공학석사 학위논문

HEVC 인코더의 Inter Prediction 연산 복잡도 감소를 위한 Reuse Buffer 구조에 대한 연구

Reuse Buffer Architecture for Reducing the Computational Complexity of Inter Prediction in HEVC encoder

2015 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

노선일

HEVC 인코더의 Inter Prediction 연산 복잡도 감소를 위한 Reuse Buffer 구조에 대한 연구

지도 교수 채 수 익

이 논문을 공학석사 학위논문으로 제출함

2015 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

노 선 일

노선일의 공학석사 학위논문을 인준함

2015 년 2 월

위 원 장 최 기 영 (인)

부위원장 채 수 익 (인)

위 원 김 수 환 (인)

초 록

Motion Estimation(ME) 은 integer ME(IME)와 fractional ME(FME)로 구성되어 있다. IME 는 모든 개별 prediction unit(PU)에 대한 integer motion vector(IMV)를 찾고, FME 는 정수 사이의 fractional pixel 을 생성한 후 개별 PU 에 대한 fractional motion vector(FMV)를 찾는다.

IME 의 초기 탐색 지점은 AMVP candidate list 로부터 선택하게 되는데, 이 리스트는 HEVC 인코더의 pipeline 구조를 고려할 때 현재 PU 의 available 한 이웃의 MV 들 중에서 선택하여 생성한다. IME 에서는 TZS 알고리즘에 의해 선택된 search point 들에 대해 sum of absolute difference(SAD) 를 이용한 low complexity RD cost(LRD cost)를 계산하고, 이들 중에서 최소 RD cost 값을 가진 search point 를 predicted motion vector 로 선택한다. FME 에서는 먼저 7 개나 8 개의 integer pixel 을 이용한 보간 필터로 fractional pixel 을 생성한 후, IMV 주변의 search point 들 중에서 sum of absolute transformed difference(SATD)로 계산한 minimum RD cost 인 것을 fractional motion vector 로 선택한다.

HM 인코더의 RDO 탐색 알고리즘은 모든 reference picture 에 대해 coding tree unit(CTU)내의 모든 가능한 CU partition 의 모든 가능한 PU partition 들에 대한 MV 를 찾게 된다. 그러므로 bi-prediction 까지 고려하면 CTU 마다 1000 개 이상의 motion vector 가 존재할 수 있다. 하나의 IMV 를 찾기 위해서, TZS 알고리즘은 평균 100 개 이상의 search point 에 대해 탐색한다. FMV 를 찾기 위해서, FME 알고리즘은 적어도 16 개 의 search point 를 탐색한다. 그러므로, SAD, SATD,

그리고 interpolation 은 HEVC 인코더의 전체 연산 복잡도의 큰 부분을 차지하고 있다.

본 논문에서는 연산 복잡도를 줄이기 용이한 reuse buffer 를 만들기 위해 SAD 연산량을 줄이고 연산 중복성은 높이는 수정된 TZS 알고리즘을 제시하였다. 더불어 SAD, SATD, 그리고 interpolation 연산 결과를 on-chip buffer 에 저장한 후 ME 수행 과정에서 재사용함으로써 중복 계산된 연산을 줄이는 구조를 제안하였다. 특히 SAD 와 SATD 의 경우, 저장된 데이터를 효율적으로 관리하기 위해 CU depth 마다 병렬적으로 처리가 가능한 cache 구조의 계층적 reuse buffer 구조를 적용하였다.

제안한 reuse buffer 구조를 사용하여 병렬 수행을 고려하여 실험한 결과, 20KB 의 on-chip 메모리를 사용하여 전체 SAD 연산량의 약 34.4%, 20KB 의 on-chip 메모리를 사용하여 전체 SATD 연산량의 18.3%, 그리고 256KB 를 사용하여 전체 interpolation 연산량의 50%를 절약하는 결과를 얻었다. 이는 SAD, SATD 및 interpolation 연산량이 전체 인코더의 약 50%를 차지하는 것을 고려할 때, 제안한 reuse buffer 를 사용하면 전체 인코더의 computational complexity 를 약 18%정도 줄일 수 있음을 의미한다. 이 때, 수정한 TZS algorithm 으로 인한 0.09%와 pipeline 구조로 인해 ME 초기 조건 변화로 발생하는 0.26%를 포함하여 0.35%의 성능저하가 발생한다.

주요어 : HEVC, Motion Estimation, Data Reuse, SAD, SATD, Interpolation

학 번 : 2013-20784

목 차

표 목차

그림 목차

제 1 장 서 론

1.1 연구의 배경

1990 년대 중반 이후 인터넷이 널리 확산되고 사용자간의 대용량 멀티미디어 데이터 저장 및 전송에 대한 요구가 높아지면서, 이를 위한 고효율의 영상 압축 기술에 대한 요구가 꾸준히 증가하게 되었다. 이러한 요구에 대응하기 위해 1990 년 대의 H.261 코덱을 시작으로 ITU-T 및 MPEG 그룹에 의해 비디오 코덱에 대한 표준화가 진행되어 왔다. 최근까지 널리 사용된 압축 표준은 두 그룹에 의해 공동으로 개발된 H.264/AVC 코덱으로, DMB, Blue-Ray, IP TV 등에 폭넓게 사용되어 왔다.

그러나, 4K 이상의 고화질 영상(1920x1080 의 4 배 해상도)에 대한 요구가 점점 증가하고 있고, 더불어 스마트폰 과 같은 모바일 기기의 보급으로 무선 통신 환경에서도 고화질의 멀티미디어 데이터 전송이 필요하게 되었으나, 기존의 H.264/AVC 의 압축성능으로는 만족할 만한 품질을 얻기 힘들게 되었다. 이러한 문제를 해결하기 위해서 ITU-T 와 MPEG 그룹의 공동 영상 표준화 그룹인 Joint Collaborative Team on Video Coding (JCT-VC)는 2 배 이상의 압축률을 가지면서도 동일한 정도의 화질을 제공하는 차세대 영상 표준인 High Efficiency Video Coding(HEVC)의 표준화 작업을 진행하였으며, 2013 년 1 월 25 일 표준안을 승인하였다.

HEVC 는 압축 효율을 높이기 위해 기존의 H.264/AVC 에서 사용하던 압축기술을 확장하여 사용하였다. H.264/AVC 에서는 코딩의 기본단위가 16x16 화소 단위의 단일 계층의 매크로블록(macroblock;MB)이었던 데

반해, HEVC 에서는 코딩의 기본단위로 64x64 화소 단위의 여러 계층의 Coding Tree Unit(CTU) 구조를 도입하여 사용하고 있다. 64x64 크기의 CTU 내부는 다시 32x32, 16x16, 8x8 크기의 여러 Coding Unit(CU)로 분할될 수 있다. 예측 단위(Prediction Unit;PU)는 16x16 MB 내에서 4x4 크기까지 분할되는 H.264/AVC 에 비해 다양한 크기의 CU 내에서 64x64 부터 4x4 까지 다양한크기로 분할되어 움직임을 예측할 수 있게 되었다. 인트라 예측모드 또한 기존 9 개의 모드에서 35 개로 증가하였으며, 8x8 및 4x4 크기만 지원되었던 변환 블록(Transform Unit; TU)또한 32x32 부터 4x4 까지 다양하게 지원하고 있다.

이와 같이 여러 계층의 다양한 예측모드 및 분할된 블록에 대해서 복원 이미지의 열화의 정도가 기존과 유사하면서도 적은 비트율의 비트스트림을 생성하여 고효율의 압축 성능을 달성하기 위해서는 당연하게도 HEVC 인코더의 복잡도를 증가시킬 수 밖에 없다. HEVC 인코더는 기존 H.264/AVC 인코더의 복잡도 대비 대략 5 배 정도의 연산 복잡도의 증가가 예상되고 있다[1].

그러므로, 본 논문에서는 인코더의 연산 복잡도가 큰 부분을 파악하고, 그 중에서도 inter prediction 부분에 대해서 기존에 수행된 연산을 저장한 후 재사용하여 연산 복잡도를 줄이는 방법을 제안한다.

1.2 연구의 내용

HEVC 의 Inter prediction 은 이전에 코딩 된 참조 영상으로부터 움직임을 예측하여 현재의 블록을 위한 prediction sample 을 생성하는 과정을 말한다.

CU 단위로 분할된 데이터는 CU 및 PU 의 Syntax Element(SE)에 의해 SKIP mode, Merge mode 및 AMVP mode 로 나뉘며 각각 candidate list 를 만들어서 motion vector 에 대한 예측을 수행한다.

AMVP mode 의 경우, 리스트에서 선택된 motion vector predictor 를 기반으로 2Nx2N, 2NxN, Nx2N, nLx2N, nRx2N, 2NxnU, 2NxnD 등의 분할 가능한 PU 에 대해 움직임 예측(Motion Estimation; ME) 연산을 수행한다. ME 는 Integer ME(IME)와 Fractional ME(FME)로 구분되는데, IME 는 가장 좋은 것을 찾기 위해 여러 장의 reference frame 을 이용하고, FME 는 IME 에서 선택한 reference frame 에 대해 half-pel 및 quarter-pel 연산이 수행된다.

IME 연산에서는 여러 장의 참조 화면 중에서 하나를 선택하며, 선택된 화면에 있는 여러 개의 search point 들 중의 하나를 integer motion vector(IMV)로 선택한다. 최적의 MV 를 선택하기 위하여 SAD 연산이 각 search point 에 대해 수행되며, 가장 작은 SAD 결과값을 가진 search point 가 최적의 IMV 로 선택된다.

FME 연산에서는 IME 연산 과정에서 구한 IMV 를 중심으로, sub-pel 을 생성하여 보다 정확한 motion vector 의 위치를 예측한다. half-pel 과 quarter-pel 은 선택된 참조 화면에 있는 IMV 의 인접 정수 화소에 7tap 및 8tap filter 를 적용하여 생성한다. 이러한 과정을 interpolation 이라고 한다. 최적의 MV 를 선택하기 위해서 SATD 연산이 각 search point 에 대해 수행되며, 가장 작은 SATD 결과값을 가진 search point 가 최적의 MV 로 선택된다.

```
...
  LOOP CU in CTU
    LOOP PU partition
      AMVP
      FOR ref. List = 0 to 1
        FOR 0 to refIdx
          Integer ME
            SAD calculation
            Decide best RD-cost
          Fractional ME
            interpolation for sub pixel
            SATD calculation
            Decide best RD-cost
      IF bi-directional prediction THEN
        FOR iteration = 0 to 3
          FOR 0 to refIdx
            Integer ME
              SAD calculation
              Decide best RD-cost
            Fractional ME
              interpolation for sub pixel
              SATD calculation
              Decide best RD-cost
      Merge
      Decide best RD-cost among uni/bi-prediction and merge
...
```

그림 1 inter mode prediction flow in HM

그림 1 에 CTU 내의 모든 CU 들에 대해서 CU 내의 모든 PU partition 들 각각에 대해 motion vector 를 찾는 과정을 표시하였다. ME 연산은 PU 마다 수행되며, IME 일 경우는 search algorithm 에 따른 search point 각각에 대해 SAD 값을 구하게 되고, FME 일 경우 역시 정해진 algorithm 에 의해 결정된 search point 각각에 대해 SATD 값을 구하여 best position 을 선택한다. 그리고, 그 과정에서 integer pixel 들 사이의 sub-pixel 을 구하는 interpolation 연산을 수행하게 된다. 그러므로, ME 내의 SAD 연산, SATD 연산, 그리고 interpolation 연산은 반복 수행되므로 연산 복잡도가 매우 큰 과정이다.

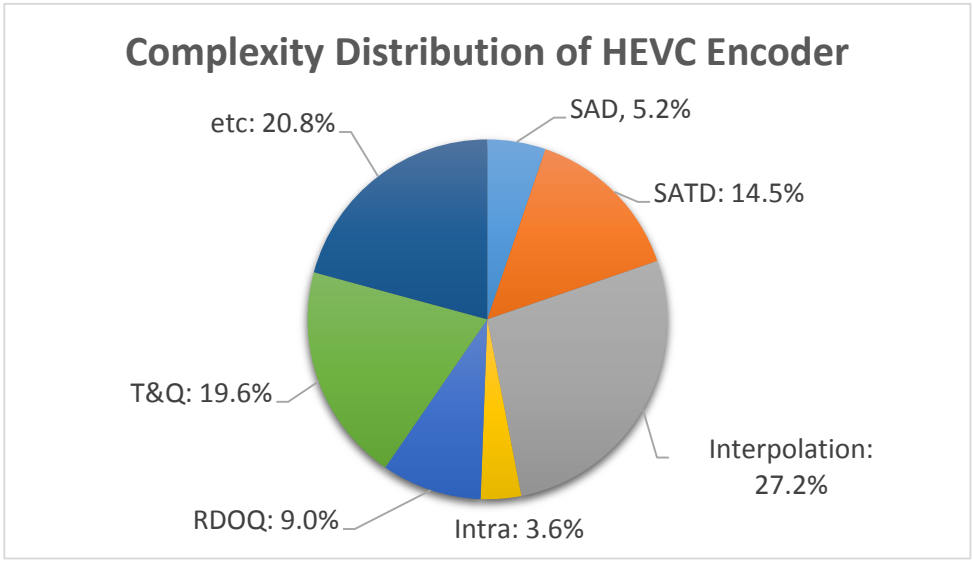


그림 2 Computational Complexity Distribution of HM Encoder

(1080p sequences, Random Access, QP=22,27,32,37, by GNU profiler)

그림 2 는 Kimono, PartkScene, Cactus, BasketballDrive, BQTerrace 5 개의 1080p sequence 에 대한 HM 인코더의 각 부분별 연산 복잡도를 나타내고 있다. SAD 연산은 전체 인코딩 연산의 약 5.2%, SATD 는 약 14.5%, 그리고 interpolation 연산은 약 27.2%를 차지하고 있다. [2].

이러한 연산량을 줄이기 위해서 기존에 이미 계산된 데이터를 활용하는 방법을 생각해 볼 수 있다. 하나의 CTU 내에 존재하는 여러 depth 의 CU 들은 HEVC 구조의 특성상 motion vector 를 찾기 위한 search range 의 범위가 유사하다. 그리고, MV 를 찾기 위해 매우 많은 search point 를 탐색하여 SAD, SATD 및 interpolation 을 구하게 되므로 이 과정에서 동일한 영역에 대한 연산이 반복되어 일어나는 경우가 많다. 그러므로, 연산을 한 번

수행한 영역에 대해 다시 연산을 수행하고자 할 때, 이전 결과를 미리 저장한 후 필요할 때 사용하면 반복되는 연산을 줄일 수 있다.

본 논문에서는 각 PU 를 4x4 단위 block 으로 나누고, 이들 block 에 대한 SAD, SATD, 및 interpolation 연산을 depth 별로 각각 저장하고 재사용하는 reuse buffer 구조를 사용하는 방법을 제안한다. 그러나 Search Range 내의 모든 SAD/SATD/Interpolation 값에 대해서 저장하기 위해서는 매우 큰 메모리가 필요하기 때문에 현실적으로 불가능하다. 그러므로 재사용 율은 크게 손해보지 않으면서 buffer 의 크기를 줄일 수 있도록 cache 와 유사한 구조의 reuse buffer 구조를 도입하여, 연산을 재사용하는 비율과 이 때 필요한 on-chip buffer 의 크기를 비교하였다.

1.3 선행 연구

기존 연구에서 ME 의 SAD/SATD/Interpolation 연산의 양을 줄이는 방법은 기본적으로 SOC 에서 전력 소모를 줄이는 방법중의 하나로 제안되고 있다. ME 의 전력 소모를 줄이기 위한 방법은 보통 두 가지로 나뉘는데, 하나는 외부 memory 에서 reference pixel 을 가져오기 위한 전력, 즉 memory access power 를 줄이는 것이고, 또 하나는 SAD/SATD/Interpolation 값을 계산하는 processing element 의 사용 전력, 즉 computational power 를 줄이는 것이다.

논문 [3]과 [4]에서는 외부 memory 에 대한 접근 횟수를 줄여서 IME 의 전력 소모를 줄이는 방법에 대해 기술하고 있다. 일반적으로 HW 를 구현할 경우 최적의 integer motion vector 를 찾고자 할 때, 모든 탐색영역에 대해

SAD 값을 계산하는 full search(FS) algorithm 을 선택한다. 이는 FS 방법의 성능이 좋기도 하지만 HW 구현이 용이하기 때문인데, 대신 reference pixel data 를 가져오기 위한 메모리 접근이 빈번하게 발생하게 된다. 그러므로, 이러한 단점을 보완하고자 16x16 크기의 reference pixel 데이터를 buffer 에 임시로 저장하여 SAD 계산에 사용하고, 그 다음 영역에 대해 계산하고자 할 때 겹쳐지는 영역에 대한 데이터는 재사용하는 방법으로 메모리 접근 횟수를 줄이고 있다[3, 4]. 이 방법을 사용하면 pixel 순서대로 차례차례 데이터를 가져오는 FS 방식을 고려할 때 매우 효율적으로 메모리 접근을 줄일 수 있다.

한편, computational power 를 줄이는 것은 두 가지 관점에서 이루어진다. 하나는 SAD 의 계산 값을 재사용하여 SAD 연산을 줄이는 것이고, 다른 하나는 search algorithm 을 FS 가 아닌 fast algorithm 을 도입하여 search point 의 수를 적게 함으로써 연산 양을 줄이는 것이다. SAD 계산 값을 재사용하는 것은 작은 영역들에 대한 SAD 연산 값 각각을 모두 더한 것은 작은 영역들을 합한 전체 영역에 대해 계산한 SAD 값과 동일하다는 특성을 이용한 것으로 [5-8]에서 이러한 특성을 이용하고 있다. Search point 의 수를 줄여서 SAD 연산 양을 줄이는 3 step search (3SS) [9], 4 step search (4SS) [10], 그리고 diamond search (DS) [11] 등의 알고리즘이 제안되었으나 이는 fixed block size 를 위해 고안된 것이므로 HEVC 에서 사용하기에는 효율적이지 않다. HEVC 에서는 test zone search (TZS) [12] 라는 알고리즘을 사용하여 variable block size(VBS)에서도 motion vector 를 효율적으로 찾을 수 있도록 하였다.

한편, SATD 와 interpolation 연산을 줄이는 방법은 integer motion vector 를 결정할 때 탐색했던 위치 중에서, cost 값이 최소인 지점의 half

pixel 데이터 및 SATD 값을 저장한 후, integer motion vector 가 같을 경우 이를 재사용하는 연구가 있었다[13, 14]. 이 논문에는 이 방법으로 약 50KB 의 SRAM 을 사용하여 65%의 interpolation 연산과 SATD 연산을 절약할 수 있었다고 한다. 이는 CU 의 partition 수가 HEVC 보다 적었던 H.264/AVC codec 에 적용된 것으로, partition 수가 많아서 일치하는 integer motion vector 비율이 더 적은 HEVC 에 적용하기에는 무리가 있다.

본 논문에서는 위에서 언급한 방법 중에서 SAD/SATD/Interpolation 의 연산 량을 줄여서 computational power 를 줄이는 방법을 제안하고자 한다. 이를 위해서 기존에 계산했던 내용을 내부 buffer 에 저장했다가 필요할 때 사용하여 반복적인 불필요한 계산을 줄이고자 한다. 기존 연구와의 차이점은, SAD 일 경우 데이터 재사용이 용이한 full search 가 아닌 TZS 알고리즘에 적용하여 데이터 재사용이 쉽지 않은 fast search 에도 reuse buffer 를 사용한 SAD 연산 재사용이 효과적임을 제시한 것이다. SATD 또한 cache 구조의 reuse buffer 를 사용함으로써 데이터 관리를 효과적으로 수행하여 일부 값이 아닌 최근에 사용한 SATD 값을 저장하여 연산값의 재사용율을 높이는데 기여하였다. Interpolation 역시 일부 값을 저장하는 것이 아니라, CTU 단위의 interpolation 값을 미리 저장한 후 재사용하는 방법론을 제시하여 interpolation 연산의 재사용율을 높이는 방법을 제시하도록 할 것이다.

1.4 논문의 구성

본 논문의 구성은 먼저, 재사용의 대상이 되는 SAD, SATD 및 interpolation 연산이 수행되는 inter prediction 의 전체적인 내용 및 motion

estimation 에 대한 내용을 2 장에서 다룬다. 3 장에서는 본 연구에서 가정하는 h/w 의 pipeline 구조에 대해 설명한다. 4 장, 5 장 및 6 장에서는 SAD, SATD, 그리고 interpolation 각각에 대한 재사용 알고리즘에 대해 기술한다. 7 장에서는 이러한 알고리즘을 구현하기 위한 reuse buffer 의 구조와 알고리즘에 대해 설명하며, 해당 내용을 구현한 실험 결과를 8 장에 기술하여 효용성을 보이고자 한다. 마지막으로 9 장에서 결론을 내린다.

제 2 장 Inter Prediction Flow

2.1 Inter Mode Decision

Inter prediction mode 는 CU 의 cu_skip_flag 와 PU 의 merge_flag 의 조합에 따라 각각 SKIP/Merge mode 와 AMVP mode 로 구분될 수 있다. SKIP/Merge mode 는 motion vector candidate list 를 구성하는 방법이 같으며, motion vector 와 reference index, predFlag 등의 정보를 이미 코딩 완료된 공간적인 이웃 블록(spatial neighbor block)과 시간적인 이웃 블록(temporal neighbor block)의 것을 그대로 받아서 활용한다. 이 때, 이웃 블록의 motion vector 정보를 이용하여 최대 크기 5 개의 merge candidate list 를 생성하며, 이 중에서 하나를 고르는 정보는 PU 의 merge_idx 로 인코딩된다.

AMVP mode 에서는 merge 와 다르게 motion vector difference(MVD)에 대한 정보를 생성하여 인코딩하게 된다. SKIP/Merge mode 와 유사한 방식으로 spatial, temporal 이웃의 정보를 기반으로 크기 2 의 AMVP candidate list 를 구축한다. PU 의 mvp_lx_flag 값에 따라 candidate 로부터 받아온 motion vector 를 motion vector predictor(MVP)로 사용하여 최종적인 MV=mvp+mvd 연산으로 motion vector 를 구하게 된다.

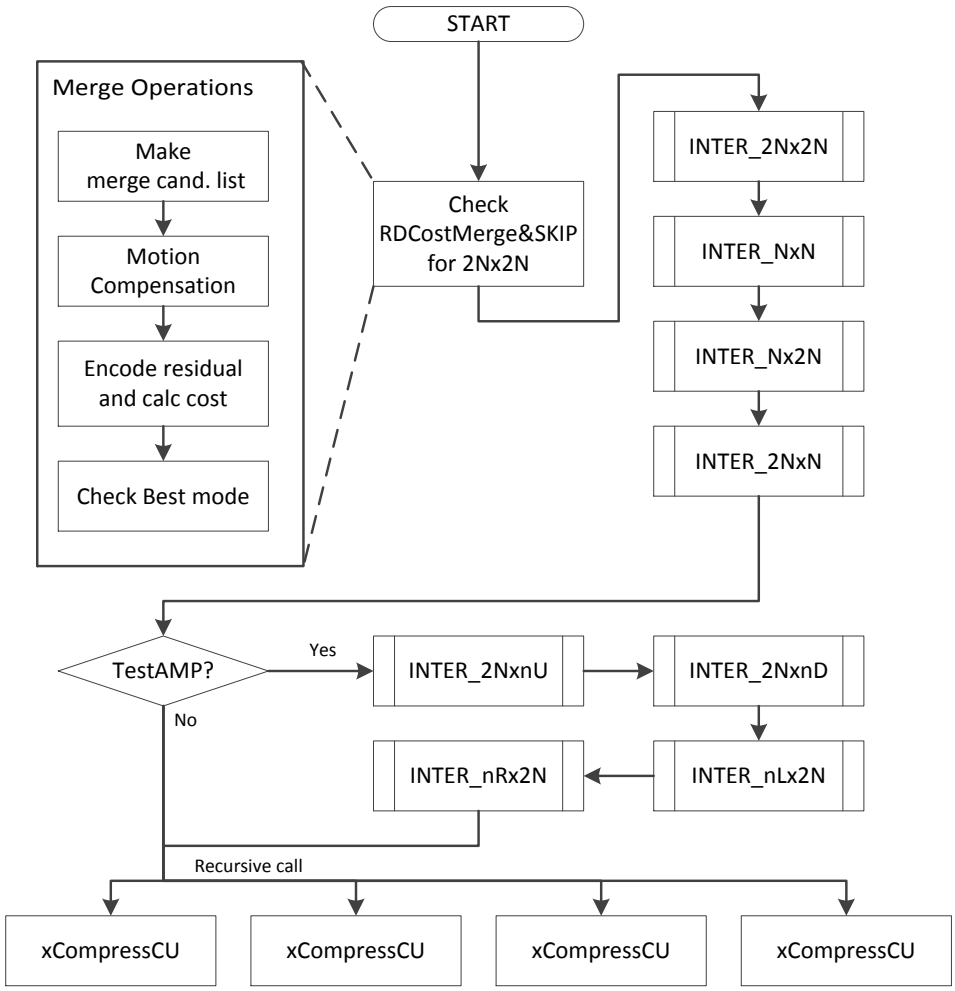


그림 3 Inter mode decision flow

그림 3 은 HM 에 구현되어 있는 inter prediction 의 전체 흐름을 나타내고 있다. 먼저, xCheckRDCostMerge2Nx2N 에서 skip 및 merge 에 대한 RD-cost 를 먼저 계산한다. 그리고 AMVP mode 로 들어와서 2Nx2N, NxN, Nx2N, 2NxN, 2NxnU, 2NxnD, nLx2N, nRx2N 등의 PU mode 순서대로 RD-cost 를 계산한다. 이 때, AMP(Asymmetric Motion Partition) 조건에 따라 비 대칭적인 partition 에 대해서는 RD cost 계산을 넘어갈 수도 있다.

이와 같이 계산된 RD-cost 중에서 최적의 RD-cost 를 갖는 PU 를 현재 CU 에서 최적의 PU 로 선택한다.

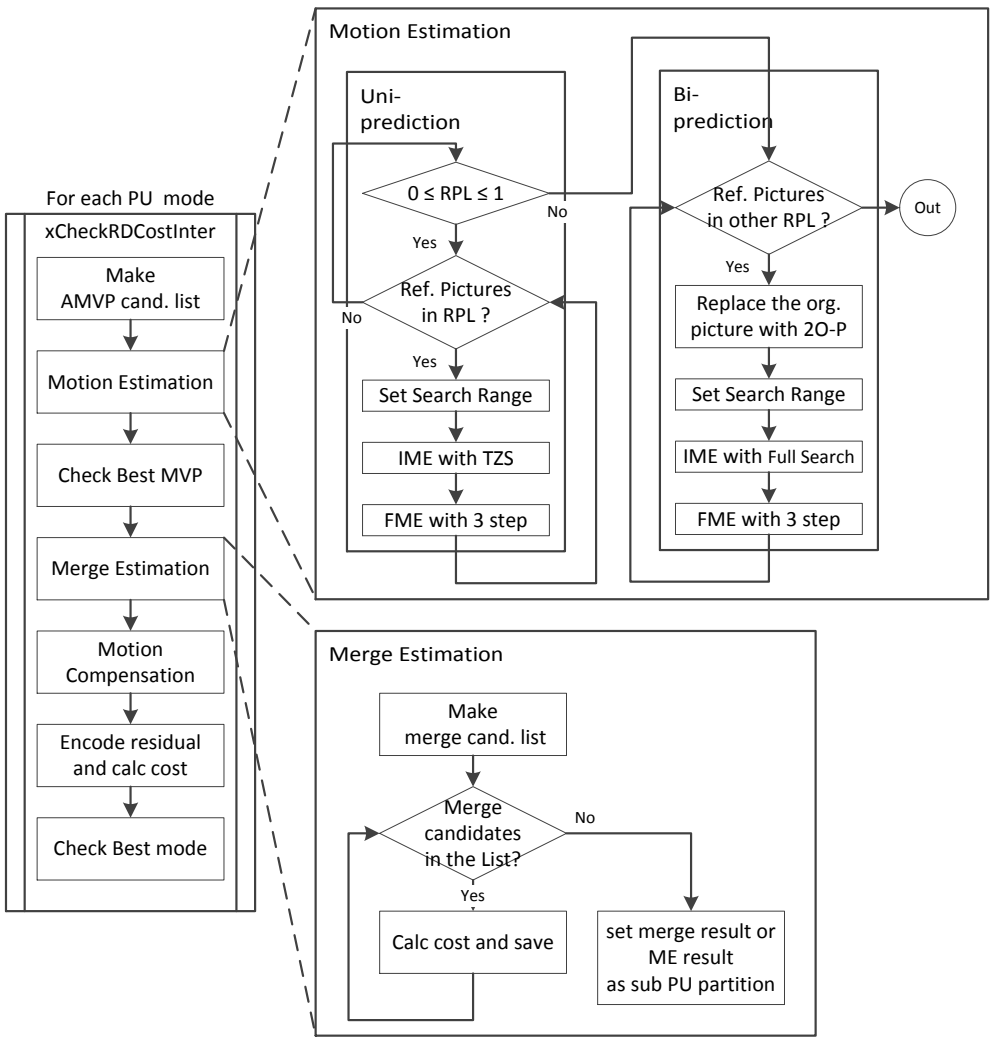


그림 4 Inter prediction process for each PU mode

그림 4 는 그림 3 에서 PU mode 대로 순서대로 수행하는 inter prediction process 에 대해 좀 더 상세하게 표현한 그림이다. 먼저 motion estimation 을

수행할 때의 기준점을 선택해야 하므로, AMVP candidate list 를 생성하고, 이들 중에서 SATD 값이 작은 것을 선택하여 motion vector predictor(MVP)로 결정한다. Motion Estimation 과정에서는 MVP 를 기준점으로 하여 지정된 탐색 알고리즘으로 reference picture 의 여러 지점을 탐색하여 최적의 motion vector 를 선택하게 된다.

먼저, uni-prediction 의 경우, RPL 에 있는 reference picture 에 대해 모두 IME 연산을 수행하여 reference picture 를 찾고, 해당 picture 의 predicted integer pixel 에 대해 interpolation 을 수행한 후 FME 연산으로 sub pixel 을 찾는다. 이 때, IME 에서는 TZS algorithm 을 사용하여 motion vector 를 구하기 위한 integer pixel 을 찾으며, FME 에서는 half-pel search 와 quarter-pel search 의 2 step search algorithm 으로 predicted pixel 을 찾는다. IME 는 SAD, FME 는 SATD 연산의 distortion 및 PU partition 과 motion vector 의 rate 을 이용하여 cost 를 구한다.

한편, bi-prediction 의 경우 uni-prediction 에서 찾은 picture, 즉 최소 cost 가 존재하는 reference picture P 를 이용하여 original picture(O)를 (2O-P)로 변경한 후, IME 와 FME 연산을 수행하여 cost 를 구한다. 이 때, IME 에서는 TZS 대신 좁은 영역(HM 의 common test condition 의 경우 ±4)을 Full search 하여 motion vector 를 선택하게 된다. 이후에는 마찬가지로 interpolation 수행 후 2 step search 로 sub pixel 을 찾는 FME 과정이 진행된다.

이런 과정을 거쳐서 선택된 predicted pixel 과는 별개로, merge estimation 도 진행되며, 이들 과정을 거쳐서 선택된 predicted pixel 들끼리 다시 RD-cost 를 비교하여 현재 CU 에서 최적 partition mode 를 선택하게 된다.

2.2 Motion Estimation

2.2.1. IME

HM 에서는 Integer Motion Vector 를 찾기 위해서 Test Zone Search(TZS) algorithm 을 채택하고 있다[12].

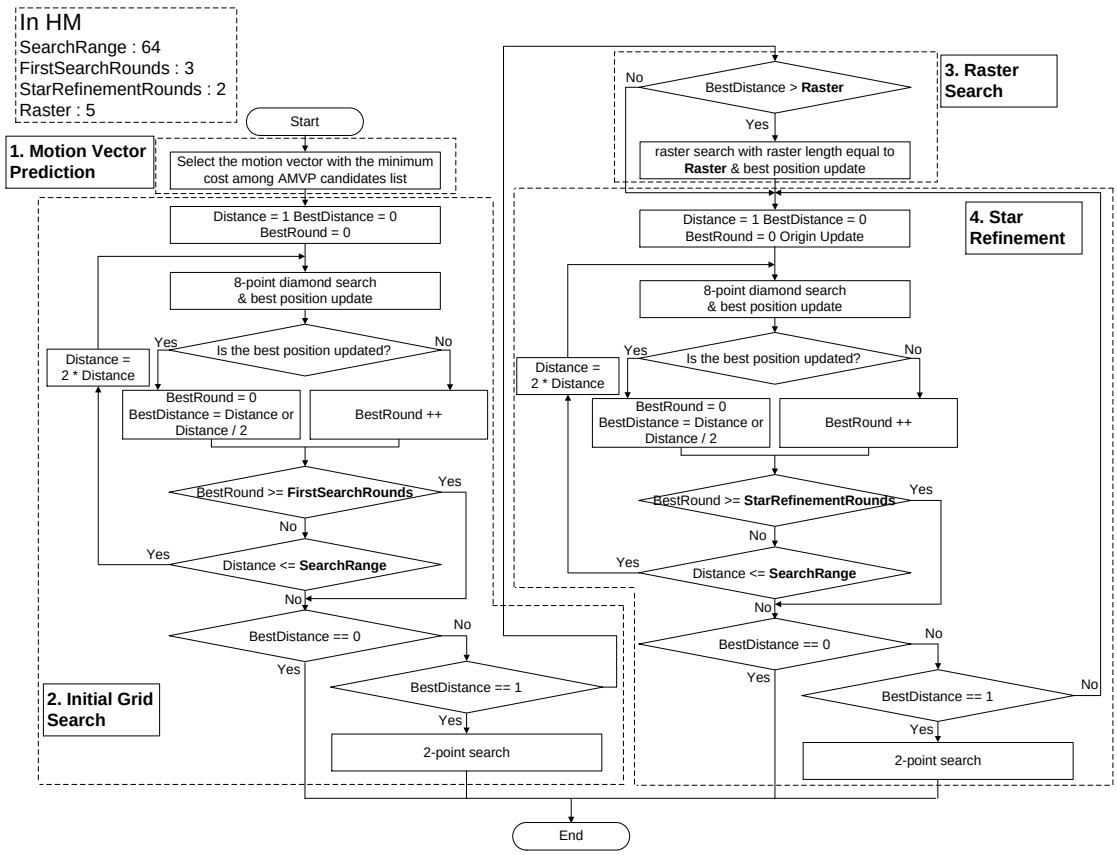


그림 5 TZS algorithm flow in detail

그림 5 는 TZS algorithm 의 전체 내용을 나타낸 순서도이다. TZS algorithm 은 Search Range 영역의 모든 점을 Full Search 로 계산하여 motion vector position 을 찾는 대신, Full Search 방식의 엄청난 연산량을 줄이기 위하여 먼저 grid search 로 단계별로 4 개 혹은 8 개씩의 search

point 들만 계산하도록 한다. Sampling 한 search point 에 대해 계산하여 best position 을 찾되, 그 위치가 기존에 설정한 범위(HM 에서는 5 로 설정) 밖에 있다면 SR 전체에 대해 raster search 로 균등하게 sampling 하여 최소 위치에 대한 경향성을 파악한 다음, 그 위치를 다시 시작점으로 하여 star refinement 로 세밀하게 계산하여 motion vector 의 값을 가져오도록 하는 것이 기본 흐름이다. 순서도의 좌 상단에 HM 에서 설정한 TZS algorithm 에 대한 parameter 값에 대해 표시하였다. TZS algorithm 은 위 그림에서처럼 4 개의 부분으로 분류되며, 아래에서 하나씩 살펴보도록 하겠다.

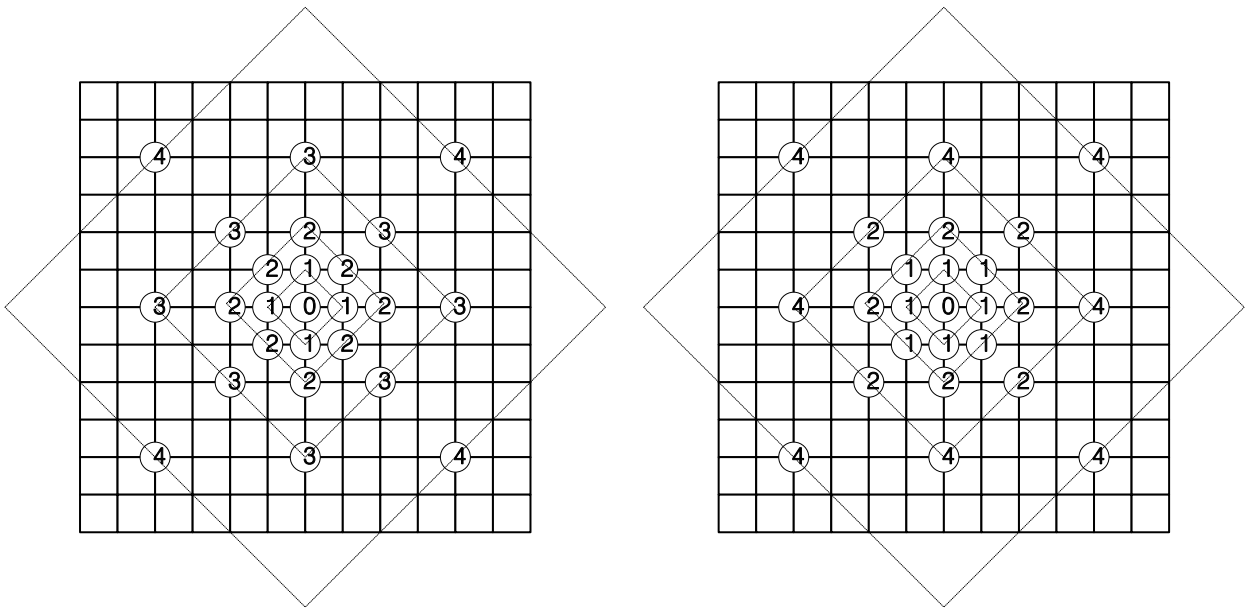
1) Motion Vector Prediction

AMVP list 에서 cost 가 가장 작은 motion vector candidate 를 하나 선택한다. 선택된 motion vector 는 zero vector 와 비교하며, 더 작은 값을 가진 지점을 선택하여 이후 TZS algorithm 의 start position 으로 사용된다.

2) Initial Grid Search

그림 6 는 Initial Grid Search 의 기본 개념을 나타내고 있다. Grid Search 에는 'round' 라는 개념이 있는데, 하나의 diamond 모양에 있는 search point 의 집합을 의미한다. 그림 6 의 (a) 에 있는 같은 숫자끼리의 집합이 같은 round 라고 생각하면 된다. grid search 에서 round 가 하나 증가했다는 것은, 같은 round 에 속한 모든 search point 에 대해 SAD 연산을 통한 cost 계산이 완료되었음을 의미한다. 한편, round 와 또 다른 개념인 'distance' 도 있다. 그림 6 의 (b)에 나타나 있는데, 이는 처음 위치로부터의 거리를 의미한다. Round 2 이상에서는 같은 round 라고

하더라도 diamond 형태의 꼭지점에 있는 search point 의 distance 와 꼭지점 사이에 있는 search point 의 distance 가 같지 않다.



(a) Rounds of 8-point diamond search (b) Distance of 8-point diamond search

그림 6 basic concept of initial grid search

각 round 에서는 모든 search point 에 대해 SAD 값을 계산하여 그 값이 기존 best SAD 값보다 작으면, SAD 값과 그 때의 x, y 좌표, 그 때의 distance 를 update 하여 새로운 best position 으로 선정한다. 또한 이 때 round 의 값을 0 으로 초기화한다. 만약 한 round 의 모든 search point 에 대한 SAD 값이 기존 best SAD 보다 작지 않으면 update 되는 position 이 없으므로 round 횟수만 증가하고 그 다음 round 에 대한 연산을 시작하며, 미리 설정한 round 횟수에 도달하면 탐색을 마치게 된다. 즉, 미리 지정한 round 만큼 탐색하는 동안 기존 best SAD 보다 작은 값이 나오지 않으면 gird

search 를 종료게 되는 것이고, 만약 작은 값이 나오면 이 위치를 best position 으로 선택하면서 round 를 0 으로 하기 때문에 다시 round 를 시작하는 동안 distance 를 넓히면서 Search Range 범위 내에서 보다 더 좋은 값을 탐색할 수 있게 되는 것이다.

3) 2-point Search

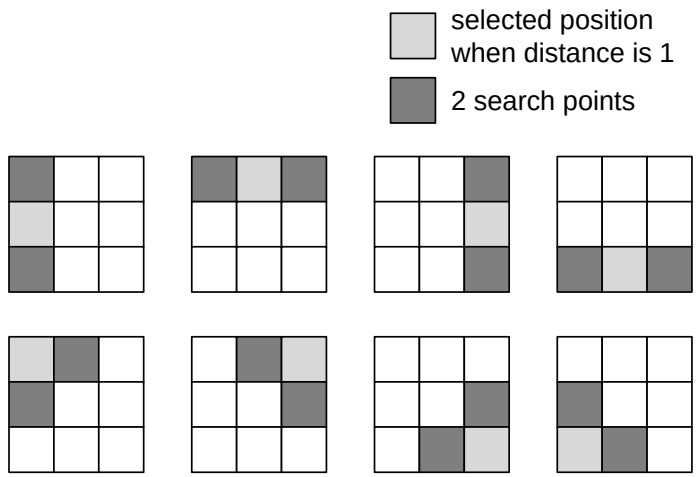


그림 7 2-point search

Grid search 의 마지막 단계에서, best distance 의 값이 0 인지 혹은 1 인지 체크하는 부분이 있다. 이는 그 값이 0 이면 정확한 motion vector 를 찾았음을 의미하므로 TZS 과정을 종료함을 의미한다. 그 값이 1 이면 그림 6(a)의 1 round 의 점을 motion vector 로 찾았다는 의미인데, 여기에서는 4 개의 search point 에 대한 연산만 진행되었기 때문에, 4 개의 search point 중 선택된 point 에 대해 연산을 수행하여 distance 가 1 인 지점들 중에 더 적합한 위치를 찾게 된다. 그림 7 에 연산 방법이 나와 있는데, 선택된 search point 주위로 2 개의 point 에 대해서만 추가적인 연산을 진행하여 best position 을 찾게 되며, 이 연산이 끝나면 TZS algorithm 도 곧바로 종료된다.

4) Raster Search

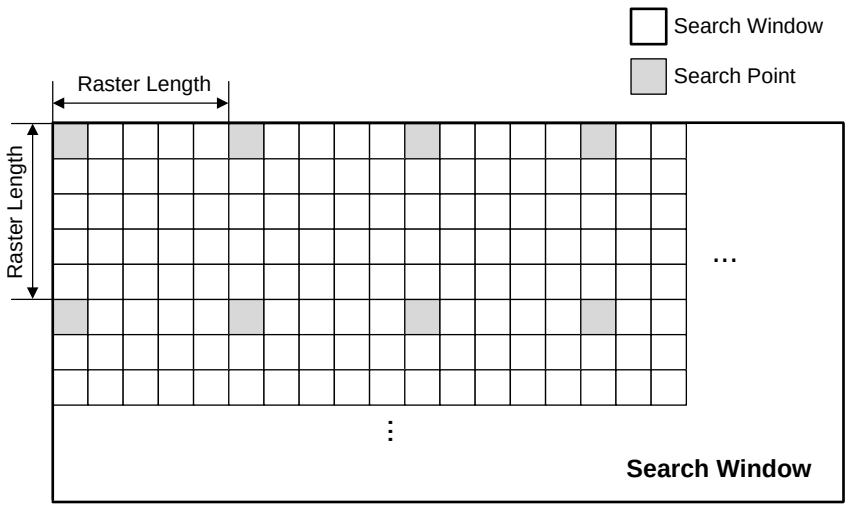


그림 8 Raster search

Grid search 를 진행한 후, best distance 가 0 이면 곧바로 종료되고 1 이면 2-point search 이후 종료되지만, 그보다 더 큰 값이면 다시 탐색 작업을 시작한다. 만약 grid search 에서 찾은 best distance 가 HM 의 config file 에서 설정한 Raster Search Length 보다 크다면, 보다 넓은 범위에서 적합한 motion vector 를 찾을 수 있다는 뜻이므로 raster search 를 시작한다. Raster search 는 Search Range 의 모든 영역에 대해서 Raster Search Length 단위로 sampling 하여 이들 지점에 대해서만 SAD 값을 계산하여 비교한다. 그림 8 에 있는 Search Point 지점이 이에 해당한다.

5) Star Refinement

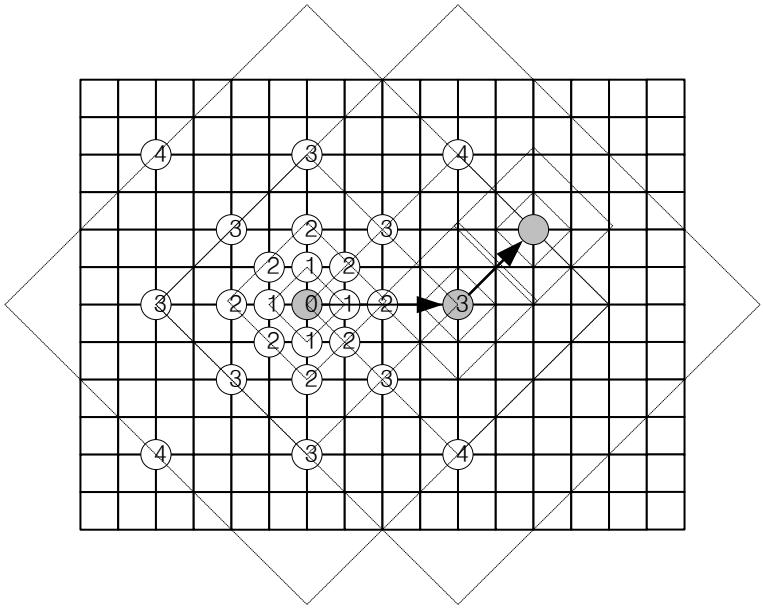


그림 9 Star refinement

Raster search 후 마지막 과정은 그림 9 에 있는 star refinement 과정이다. 여기까지 왔다는 것은, rater search 를 진행했다면 distance 5 단위로 sampling 하여 계산한 값들 중에 가장 작은 값을 motion vector 로 가지고 있다는 뜻이고, rater search 를 진행하지 않았다고 하더라도 적어도 best distance 가 2 이상이라는 뜻이다. 그러므로, star refinement 를 통해서 좀 더 최적의 vector 를 찾고자 하는 것이다. 기본적인 과정은 grid search 와 유사하지만, search 의 시작점을 해당 round 에서 찾은 best position 으로 옮긴다는 점에서 차이가 있다. 이 과정에서 최종적으로 best distance 가 0 이 되면 곧바로 종료하고, 1 이 되면 2-point search 과정을 통해서 다시 best position 을 찾은 후 TZS 알고리즘을 종료하게 된다.

2.2.2. FME

FME 는 IME 에서 구한 integer motion vector 를 기준으로 좀 더 세밀하게 sub-pixel 단위의 motion vector 를 구하는 과정이며, 세부적으로는 sub-pixel 계산, residue 생성, 그리고 hadamard transform 계산으로 구성된다.

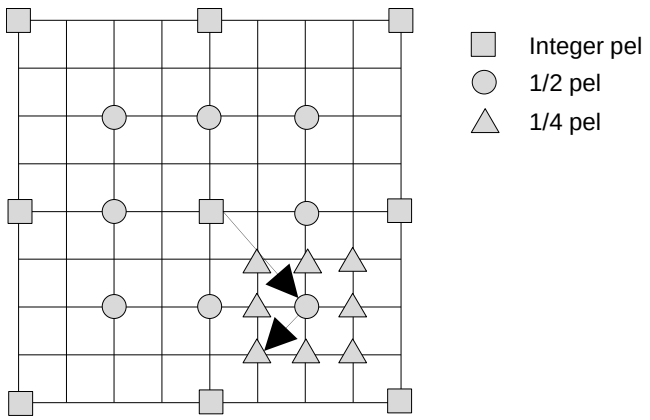


그림 10 FME search flow

그림 10 은 IME 에서 선택된 integer MV 를 기준으로 fractional motion vector 를 찾는 과정을 나타내고 있다. 모든 reference frame 들의 모든 search point 들 중에서 가장 좋은 cost 로 선택된 integer point 주변 8 개의 half-pixel 에 대해서 원본 integer pixel 과의 distortion 을 SATD 로 계산한다. 이들 중에서 cost 가 가장 작은 것을 선택한 후, 다시 해당 point 주변 8 개의 quarter-pixel 에 대해서 원본 integer pixel 과의 distortion 을 SATD 로 계산하여 cost 가 가장 작은 것을 선택하여 최종적인 predicted pixel 로 결정한다. Cost 를 계산할 때에는 distortion 뿐만 아니라 motion vector difference 및 partition mode 에 대한 bit rate 을 고려하여 구하도록 한다.

이 때, sub-pixel 은 원래 없던 값이므로, 기존에 있던 integer pixel 들의 값을 사용하여 새로운 sub-pixel 을 만들게 된다. 이러한 방법을 interpolation 이라고 하며, HEVC 표준에서는 이를 위해 7-tap 이나 8-tap 을 사용하고 있다.

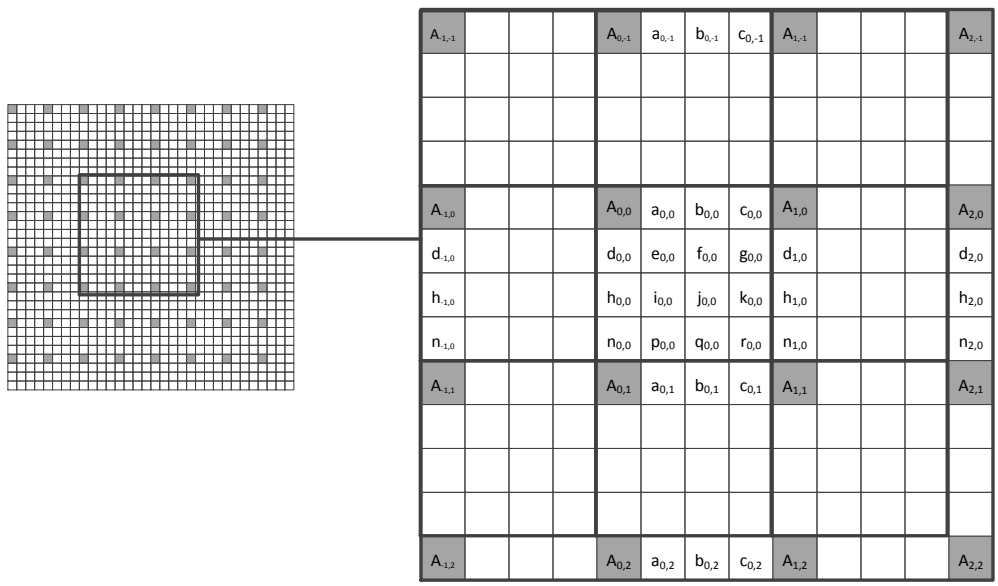


그림 11 integer pixel 위치(gray squares) 및 fractional pixel 위치

그림 11 은 interpolation 을 수행하기 위한 integer pixel(대문자로 표시)들의 위치와 그 결과 발생하는 fractional pixel(소문자로 표시)들의 위치에 대해 표시한 것이다. 그림 11 의 $a_{0,0}$, $b_{0,0}$, $c_{0,0}$, $d_{0,0}$, $h_{0,0}$, 그리고 $n_{0,0}$ 은 integer pixel 만을 사용하여 8-tap FIR filter 를 적용하여 만들 수 있는 fractional pixel 들이다. $e_{0,0}$, $f_{0,0}$, $g_{0,0}$, $i_{0,0}$, $j_{0,0}$, $k_{0,0}$, $p_{0,0}$, $q_{0,0}$, 그리고 $r_{0,0}$ 은 integer pixel 을 사용하여 만들어진 $a_{0,i}$, $b_{0,i}$, $c_{0,i}$ 을 사용하여 만들 수

있는데, 여기에서 i는 −3 부터 4를 의미한다. HEVC 표준에서 정의한 FIR filter 값을 적용하여 fractional pixel을 만드는 수식은 다음과 같다.

$a_{0,0} = (-A_{-3,0} + 4*A_{-2,0} - 10*A_{-1,0} + 58*A_{0,0} + 17*A_{1,0} - 5*A_{2,0} + A_{3,0})/64$

$b_{0,0} = (-A_{-3,0} + 4*A_{-2,0} - 11*A_{-1,0} + 40*A_{0,0} + 40*A_{1,0} - 11*A_{2,0} + 4*A_{3,0} - A_{4,0}) /64$

$c_{0,0} = (A_{-2,0} - 5*A_{-1,0} + 17*A_{0,0} + 58*A_{1,0} - 10*A_{2,0} + 4*A_{3,0} - A_{4,0}) /64$

$d_{0,0} = (-A_{0,-3} + 4*A_{0,-2} - 10*A_{0,-1} + 58*A_{0,0} + 17*A_{0,1} - 5*A_{0,2} + A_{0,3}) /64$

$h_{0,0} = (-A_{0,-3} + 4*A_{0,-2} - 11*A_{0,-1} + 40*A_{0,0} + 40*A_{0,1} - 11*A_{0,2} + 4*A_{0,3} - A_{0,4}) /64$

$n_{0,0} = (A_{0,-2} - 5*A_{0,-1} + 17*A_{0,0} + 58*A_{0,1} - 10*A_{0,2} + 4*A_{0,3} - A_{0,4}) /64$

그림 12 integer pixel에 대한 수평/수직 interpolation

$e_{0,0} = (-a_{0,-3} + 4*a_{0,-2} - 10*a_{0,-1} + 58*a_{0,0} + 17*a_{0,1} - 5*a_{0,2} + a_{0,3}) /64$

$i_{0,0} = (-a_{0,-3} + 4*a_{0,-2} - 11*a_{0,-1} + 40*a_{0,0} + 40*a_{0,1} - 11*a_{0,2} + 4*a_{0,3} - a_{0,4}) /64$

$p_{0,0} = (a_{0,-2} - 5*a_{0,-1} + 17*a_{0,0} + 58*a_{0,1} - 10*a_{0,2} +$

$4*a_{0,3} - a_{0,4}) /64$

$f_{0,0} = (-b_{0,-3} + 4*b_{0,-2} - 10*b_{0,-1} + 58*b_{0,0} + 17*b_{0,1} - 5 * b_{0,2} + b_{0,3}) /64$

$j_{0,0} = (-b_{0,-3} + 4*b_{0,-2} - 11*b_{0,-1} + 40*b_{0,0} + 40*b_{0,1} - 11*b_{0,2} + 4*b_{0,3} - b_{0,4}) /64$

$q_{0,0} = (b_{0,-2} - 5*b_{0,-1} + 17*b_{0,0} + 58*b_{0,1} - 10*b_{0,2} + 4*b_{0,3} - b_{0,4}) /64$

$g_{0,0} = (-c_{0,-3} + 4*c_{0,-2} - 10*c_{0,-1} + 58*c_{0,0} + 17*c_{0,1} - 5*c_{0,2} + c_{0,3}) /64$

$k_{0,0} = (-c_{0,-3} + 4*c_{0,-2} - 11*c_{0,-1} + 40*c_{0,0} + 40*c_{0,1} - 11* c_{0,2} + 4*c_{0,3} - c_{0,4}) /64$

$r_{0,0} = (c_{0,-2} - 5*c_{0,-1} + 17*c_{0,0} + 58*c_{0,1} - 10*c_{0,2} + 4*c_{0,3} - c_{0,4}) /64$

그림 13 sub pixel에 대한 수직 interpolation

제 3 장 가정하는 HEVC 인코더의 pipeline 구성

3.1 가정하는 pipeline 의 구성

HEVC HW 인코더는 아래 그림처럼 prediction stage, transform & full RDO stage, reconstruction & in-loop filter stage, 그리고 entropy coding stage 의 4 단계 CTU 단위 pipeline 으로 구성된다고 가정한다.

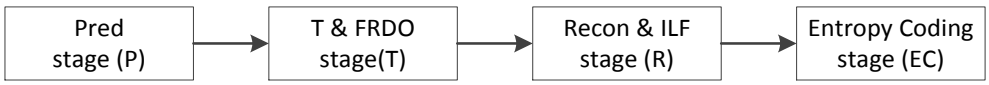


그림 14 가정하는 H/W의 pipeline 구조

또한, prediction stage 와 T&FRDO stage 는 CU depth 별로 각각 서로 다른 pipeline 크기를 가지며, PU partition 단위로 순차적으로 수행하되, 주어진 연산 unit 들로 최대한 병렬적으로 수행하는 것을 가정하였다. 또한 크기가 작은 CU depth 의 T stage 4 개가 진행될 때마다 크기가 큰 CU depth 의 T stage 와 sync 한다.

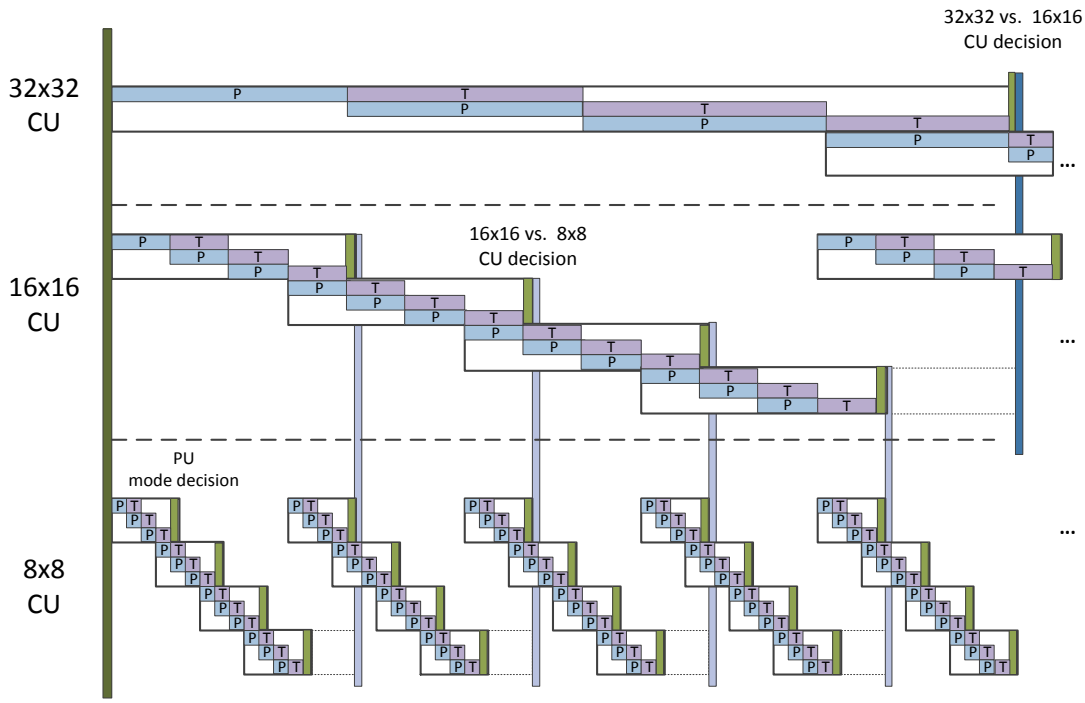


그림 15 가정하는 pipeline 구조

(각 CU 내의 P stage 와 T stage 의 조합 세 쌍은 각각 2Nx2N, 2NxN, 그리고 Nx2N PU partition 을 의미한다.)

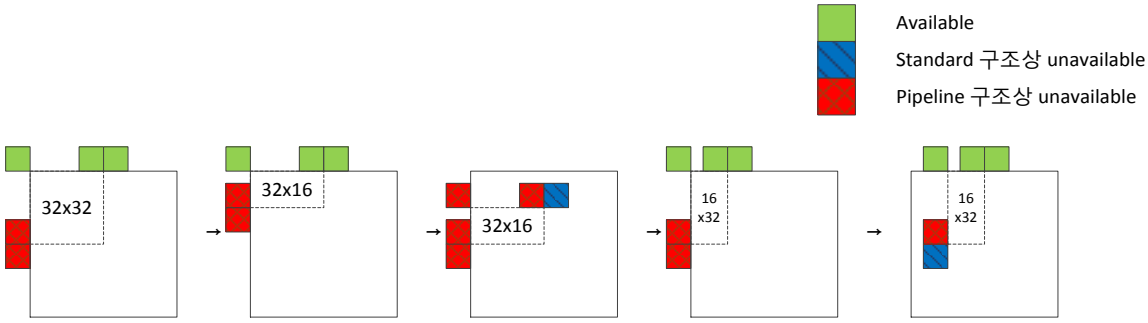
그림 15 는 위에서 설정한 가정을 만족하는 pipeline의 구조를 표현한 것이다. 각 CU depth에서는 각각 서로 다른 pipeline 구조를 가지고 있으며, 각 CU depth에 속한 PU partition들은 순차적으로 실행됨을 나타내고 있다. 하나의 CU에 대해서 2Nx2N, 2NxN, Nx2N 등의 PU partition의 P stage와 T stage가 실행된다. 하나의 CU에 대해 각 reference picture들마다 PU partition들의 P stage와 T stage가 끝나면 이들 중에서 cost가 가장 작은 것을 선택하는 decision 과정이 수행되어 최적의 PU partition 을 선택한다. 하나의 CU depth에 대해 위의 과정을 4번 반복하여 각 CU 영역에 대한 partition mode를 결정하면, 상위 CU depth에서 선택한 상위 CU의 partition

mode와 비교하여 더 작은 cost의 partition mode를 선택하여 CU partition을 결정하게 된다.

각 CU depth의 pipeline stage는 PU mode decision과는 상관없이 순차적으로 진행되게 할 수 있다. 그러므로, 하나의 CU 내의 마지막 PU partition인 Nx2N에 대한 prediction이 끝나면, 해당 CU의 decision process와는 상관없이 다음 CU의 첫 번째 partition인 2Nx2N에 대한 prediction을 진행할 수 있다. 그림 15 의 decision process에 걸쳐서 진행되는 prediction 부분이 이를 나타내고 있다. 그러나, 이 때 prediction의 시작 지점인 MVP를 결정하기 위한 AMVP candidate list를 구성할 때, pipeline구조로 인해 선택하지 못하는 candidate이 발생하게 된다. 왜냐하면, 현재 PU에서 candidate들을 선택할 때 이전 CU의 T stage가 끝나지 않으면 CU decision이 완료되지 않아서 available한 candidate을 구하지 못하는 경우가 있기 때문이다. 그러므로, pipeline 진행 및 PU의 위치를 고려하여 unavailable한 candidate을 제외한 Pseudo-AMVP list를 구성하여 ME의 시작점을 결정하도록 한다.

3.2 Pseudo-AMVP list 구성으로 인한 BD-rate 저하

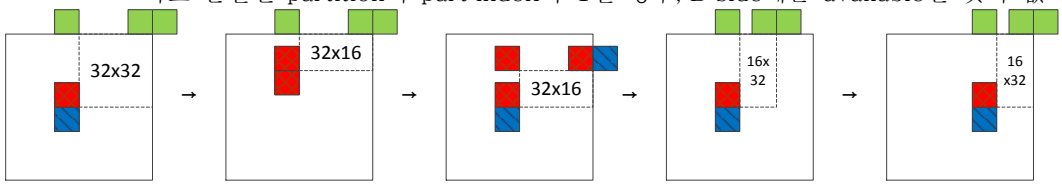
아래 그림은 pipeline 구조로 인해 HEVC standard 대비 추가적으로 unavailable한 candidate을 표시한 것이다. AMVP list를 구성할 때, 각 위치에 따라 pipeline 구조로 인해 사용할 수 없는 candidate이 어떠한 것이 있는지 따져 보고 AMVP list를 구성해야 한다. 이렇게 할 경우, 원래의 AMVP list와는 달라지게 되므로, 이를 Pseudo-AMVP candidate list (pseudo-ACL)라고 부르기로 한다.



(a) 좌상단에 위치한 32x32 CU 내의 PU partition에 대한 candidate 종류

– left side의 CU partition 결정이 되지 않아서 A side에는 available한 것이 없다.

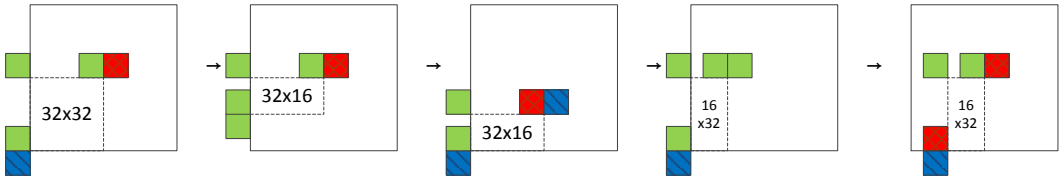
- 가로 분할된 partition의 part index가 1일 경우, B side에는 available한 것이 없다.



(b) 우상단에 위치한 32x32 CU 내의 PU partition에 대한 candidate 종류

– left side의 CU partition 결정이 되지 않아서 A side에는 available한 것이 없다.

- 가로 분할된 partition의 part index가 1일 경우, B side에는 available한 것이 없다.

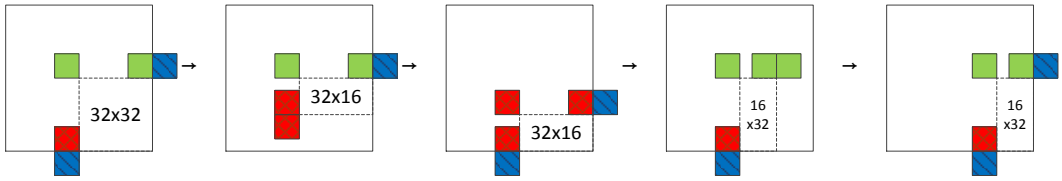


(c) 좌하단에 위치한 32x32 CU 내의 PU partition에 대한 candidate의 종류

– left side는 CU partition이 결정된 상황이므로 A side는 standard와 같이 available하다. 단, 세로 partition일 때 part index=1 이면 left candidate은 available하지 않다.

– 가로 partition이고 part index=0이면 우상단 candidate은 available하지 않다.

– 가로 partition이고 part index=1이면 상단 candidate은 available하지 않다.



(d) 우하단에 위치한 32x32 CU 내의 PU partition에 대한 candidate의 종류

– left side의 CU partition 결정이 되지 않아서 A side에는 available한 것이 없다.

가로 분할된 partition의 part index가 1일 경우, B side에는 available한 것이 없다.

그림 16 pipeline 구조로 인해 위치에 따라 추가적으로 unavailable해지는 candidates (32x32 CU에 대한 예시)

그림 16 은 위치에 따라 available 하고 unavailable 한 candidate 이 어떻게 구성되는지 표시한 그림이다. CU 가 상위 depth CU 내의 좌상단 위치일 때((a)의 경우) 왼쪽은 CU partition 결정이 완료되지 않은 상태이므로 candidate 으로 사용할 수 없다. 이것은 우상단, 우하단 위치일 때도 마찬가지이다. 예외적으로 좌하단인 (c)의 위치에서는 왼쪽의 candidate 들이 available 한데, 이는 ACL 을 구성할 때 이전 CTU 의 T stage 가 완료되도록 PE 를 할당함으로써 available 한 candidate 을 얻도록 할 수 있다. 이외에도 추가적으로 위치에 따라 part index 가 1 인 경우 unavailable 한 candidate 들이 있는데, 이를 반영하여 Pseudo-ACL 을 구성하도록 한다.

ME 는 pseudo-ACL 로 결정된 초기 조건으로 수행하게 되며, 그 결과 원래의 ACL 에 의해 결정되는 MV 와는 다른 MV^* 가 생성된다. 이미지가 인코딩될 때는 ACL 에서 선택된 candidate 의 index 와 코딩 효율을 위해 MV 가 아닌 MVD 를 rate 으로 하여 코딩하게 되는데, pseudo-ACL 에 의해 원래의 MV 가 아닌 MV^* 가 결정되어 있으므로 rate 을 구할 때는 이러한 부분을 보상해 주어야 한다.

처음 ACL 을 구하는 단계인 prediction stage 에서는 pipeline 구조에 의해 unavailable 한 candidate 이 존재하지만, transform&FRDO 단계에서는 이전 CU 에 대한 정보가 있으므로 완전한 ACL 을 구할 수 있다. 그러므로, transform&FRDO 단계에서 rate 을 구할 때, pseudo-ACL 에 의한 MV^* 와 real ACL 에 있는 candidate 들 사이에서 구한 cost 를 비교하여 결정된 candidate 으로 index 와 mvd 를 계산하여 코딩하도록 한다.

이렇게 하여 저하된 BD-rate 을 측정하였으며, RA 기준 약 0.26%가 저하되는 결과를 얻었다.

표 1 prediction stage에서 avail한 PU의 mv로 구성한 pseudo-AMVP candidate list로 시작점을 결정하였을 때 RA configuration의 BD-rate

1080p	BD rate (%)			
	Y	U	V	6:1:1
Kimono	0.26	0.45	0.16	0.26
ParkScene	0.31	0.49	0.07	0.31
Cactus	0.36	0.23	0.07	0.32
BasketballDrive	0.23	1.32	-0.09	0.29
BQTerrace	0.07	0.09	0.57	0.11
Average	0.25	0.51	0.16	0.26

제 4 장 SAD Data Reuse 알고리즘

4.1 SAD 데이터 재사용의 범위

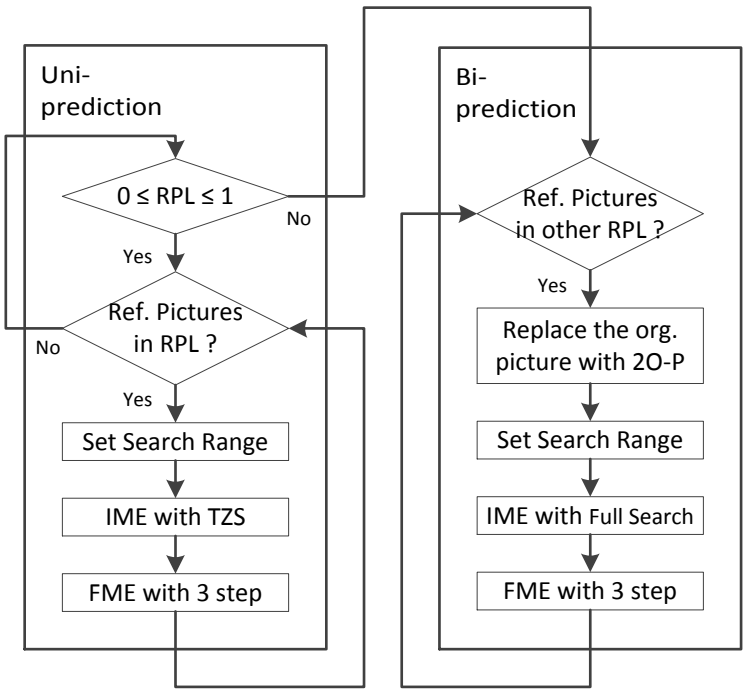


그림 17 Motion Estimation Flow

그림 17 는 앞에서 inter prediction 에 대해 설명할 때 언급했던 ME 의 flow 를 나타낸 그림이다. RPL 에 있는 reference picture 에 대해 모두 IME 연산을 수행하는 uni-prediction 의 경우, IME 연산 알고리즘으로 TZS 를 사용한다. 반면, bi-prediction 의 경우, uni-prediction 에서 찾은 picture(P)를 이용하여 original picture(O)를 (2O-P)로 변경한 후, 여기서 full search algorithm 으로 IME 를 수행하여 predicted pixel 을 구한다. 즉, bi-prediction 의 경우, original pixel 이 계속 변경되므로, 동일 영역에 대한

SAD 라 하더라도 값이 서로 달라서 재사용할 수 없는 상황이다. 그러므로, SAD 와 SATD 데이터의 재사용 범위에서 이들을 제외하도록 한다.

Bi-prediction 으로 인한 SAD 및 SATD 연산량은 전체의 약 30%이므로, 나중에 전체 연산량에서 data reuse 로 인한 연산 감소량을 계산할 때 이를 반영하도록 할 것이다.

4.2 SAD 데이터의 중복률

TZS알고리즘에서 각각의 PU들에 대한 초기 search point들은 주변 PU들의 motion vector로부터 얻어지기 때문에 같을 확률이 크다. 그러므로, CTU 내의 모든 PU partition들에 대한 SAD 값 또한 겹쳐질 확률이 크다. 이러한 확률을 측정하기 위해 PU를 4x4 block단위로 나누어 SAD값이 중복될 확률을 측정하였다.

표 2 Redundant SAD calculation ratio by 4x4 block unit considering pipeline structure

Sequences	Random Access	Lowdelay-B
Kimono	39.5%	40.3%
ParkScene	46.3%	42.0%
Cactus	44.5%	42.3%
BasketballDrive	42.0%	41.0%
BQTerrace	54.3%	51.0%
Average	43.7%	43.3%

(32frames, QP=22,27,32,37, HM 12.0 Common Test Condition[15])

현재 얼마나 많은 중복 연산이 이루어지고 있는지 파악하기 위해 HM 12.0 의 TZS search 환경에서 이를 조사하였다. 또한 앞에서 가정한 H/W 설정에 의해 CU depth 간의 SAD 는 중복되지 않는 상황을 가정하였다.

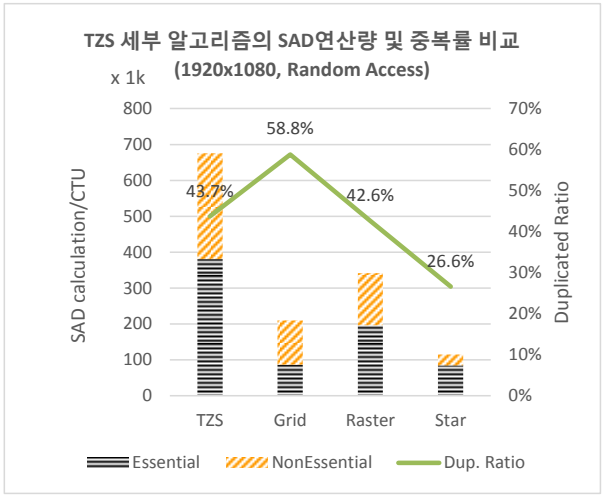
표 2 은 중복된 SAD 연산에 대해 각 이미지 sequence 별로 나타낸 것이다. 원본 이미지의 PU 를 각각 4x4 block 단위로 분할하여 이들 각각에 대해 참조 이미지의 데이터로부터 SAD 를 계산하였다. 참조하는 이미지도 같고 원본 block 과 예측 block 의 위치가 같은 4x4 block 에 대한 SAD 값을 이미 계산한 적이 있으면, 그 이후부터는 중복인 것으로 처리하였으며, 중복률 R_{DUP}는 다음과 같은 식으로 구하였다.

$$R_{DUP} = \frac{Duplicated\ SAD\ Counts}{(Essential\ SAD\ Counts + Duplicated\ SAD\ Counts)}$$

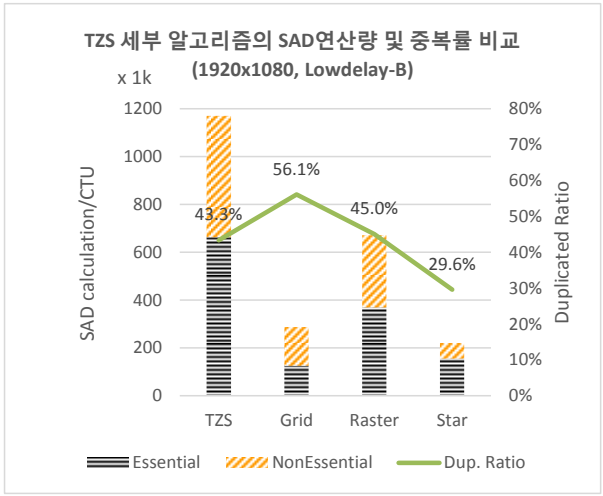
TABLE 1 에 이러한 과정으로 구한 중복률을 표시하였으며, Full HD 시퀀스 5 개에 대해 적용해 본 결과 SAD 계산의 약 43.7%가 중복 연산임을 알 수 있었다.

4.3 TZS algorithm modification

앞장에서 Test Zone Search(TZS)알고리즘에서 살펴본 바와 같이, TZS 알고리즘은 초기 predicted motion vector 를 중심으로 best position 을 찾는 Grid Search, local area 에 best position 이 없을 경우 Search Range 전체에 대한 sampling search 를 진행하는 Raster Search, 그리고 앞의 두 과정에서 찾은 결과에 대해 fine tuning 을 진행하는 Star Refinement 의 세 부분으로 나눌 수 있다. TZS 알고리즘 내의 이들 세 파트는 각각 나름의 목적이 있는 만큼 search point 를 선택하는 방법도 다르고 이에 따라 선택된 search point 들 사이의 재사용 가능한 SAD 연산의 양도 크게 다르다.



(a) Random Access



(b) Lowdelay-B

그림 18 TZS 세부 알고리즘의 중복률 및 SAD 연산량

그림 18은 TZS 세부 알고리즘의 SAD연산 중복률 및 SAD 연산량에 대해 비교한 그래프이다. 가장 왼쪽의 Original 항목은 TZS 알고리즘을 그대로 수행하였을 때의 연산량 및 중복률을 나타낸다. 그 옆의 항목들인 Grid, Raster, Star는 각각 Grid Search, Raster Search, Star Refinement 일 때의 연산량 및 중복률를 나타낸다. 왼쪽의 세로축은 CTU 당 4x4 단위의 SAD 연산량을 표시하기 위한 것이고 오른쪽 세로축은 중복률에 대한 것이다. 그리고 파란색 가로줄 부분은 반드시 수행되어야 하는 'Essential' 한 SAD 연산의 양을 표현하고 있고, 주황색 사선 부분은 'Essential' 한 SAD 연산을 다시 계산하는 중복 SAD 연산의 양을 표시하고 있다. 상단 부분의 꺾은선 그래프는 각 항목별 중복률을 표시하고 있다.

위의 각 항목에 대해서, grid 및 raster search 대비 star refinement 의 중복률이 상당히 낮으므로, star refinement 는 SAD 재사용의 대상에서

제외하도록 한다. 그리고 raster search 의 중복률이 그리 높지 않은데, 중복률을 높이기 위해 TZS 알고리즘을 변경하도록 한다.

4.3.1. Star Refinement

Star refinement 는 TZS 실행 순서로는 마지막이지만, 중복률을 높이기 위한 여러 가지 실험을 진행하려면 가장 먼저 관련 parameter 를 설정해야 한다. 이는 grid 나 raster search 에서 여러 가지 option 을 변경하여 성능 저하가 일어나더라도, star refinement 에서 보정을 해 주기 때문에 정확한 성능 비교가 힘들기 때문이다. 그러므로, star refinement 에 대해 먼저 성능을 비교한 후 round 를 고정시켜서 다른 알고리즘을 비교하기 위한 기본 조건으로 삼는다.

표 3 Star refinement의 round를 제한하였을 경우의 Bd-rate 변화 및 SAD 연산량 감소율

	Random Access					Lowdelay-B				
	6 : 1 : 1	Y	U	V	SAD 감소율	6 : 1 : 1	Y	U	V	SAD 감소율
~Round1	0.16%	0.14%	0.24%	0.19%	-15.77%	0.05%	0.06%	0.07%	0.00%	-16.9%
~Round2	0.11%	0.04%	0.33%	0.29%	-13.95%	-0.02%	-0.02%	-0.08%	0.05%	-14.6%
~Round3	0.10%	0.04%	0.24%	0.29%	-12.04%	0.01%	-0.01%	-0.03%	0.13%	-12.9%
~Round4	**0.06%**	**0.04%**	**0.15%**	**0.07%**	**-10.32%**	**0.02%**	**0.03%**	**0.17%**	**-0.16%**	**-11.0%**
~Round5	0.08%	0.06%	0.18%	0.11%	-6.77%	0.01%	-0.01%	0.09%	0.07%	-7.2%
~Round6	0.06%	0.06%	0.15%	0.00%	-3.07%	0.04%	0.03%	0.10%	0.00%	-3.3%

(1920x1080 32frames)

표 3 는 star refinement 의 실행 round 를 1 에서 6 까지로 제한하였을 경우 각각의 BD-rate 변화와 SAD 연산량의 감소율에 대해 나타낸 표이다. 실험

결과에 의하면 round 를 더 진행할수록 BD-rate 이 좋아지는 경향을 보이다가 round 4 이후로는 큰 차이가 없는 것을 볼 수 있다. Round4 이후로는 BD-rate 이 조금씩 좋아지거나 나빠지거나 하기도 하는데, 이는 star refinement 의 종료 round 가 정해진 알고리즘에 의해 동적으로 변화하는 것을 특정 round 에 종료하도록 고정시켰기 때문에 발생하는 현상인 것으로 보인다. 그러나 그 폭이 크지 않으므로 여기에서는 무시하기로 하고 star refinement 의 round 는 4 로 고정하는 것으로 한다. 이 때의 SAD 연산량은 10%정도 감소하였다.

4.3.2. Grid Search

표 4 TABLE 2 Grid Search의 실행 round에 따른 성능 변화 및 SAD연산량 증감률

	Random Access					Lowdelay-B				
	6 : 1 : 1	Y	U	V	SAD 증감률	6 : 1 : 1	Y	U	V	SAD 증감률
~Round1	1.89%	1.72%	2.35%	2.48%	-86.0%	0.39%	0.39%	0.56%	0.25%	-87.6%
~Round2	1.26%	1.11%	1.67%	1.72%	-76.4%	0.15%	0.14%	0.30%	0.05%	-80.1%
~Round3	1.23%	1.09%	1.60%	1.72%	-69.7%	0.17%	0.14%	0.38%	0.12%	-75.4%
~Round4	0.25%	0.21%	0.45%	0.32%	-24.5%	0.07%	0.07%	0.15%	0.01%	-29.1%
~Round5	**0.06%**	**0.05%**	**0.20%**	**0.00%**	**-15.0%**	**-0.08%**	**-0.04%**	**-0.19%**	**-0.24%**	**-19.3%**
~Round6	0.00%	-0.02%	0.04%	0.07%	5.8%	-0.04%	-0.03%	-0.12%	-0.05%	11.3%

(1920x1080 32frames, Star refinement round=4)

앞에서 star refinement 의 round 범위를 4 로 제한하였으므로, 이 조건아래에서 grid search 에 대해 round 횟수를 제한하면서 BD-rate 와 SAD 연산량의 관계에 대해 실험하였다. 표 4 은 star refinement 의 탐색 조건을 round4 로 고정한 상태에서 grid 탐색의 round 를 1 에서 6 까지

변경하면서 실험한 것으로, TZS 알고리즘을 수정하지 않은 결과와 비교한 값이다. Grid 탐색은 round 1 에서 3 까지는 고정적으로 실행이 되어야 하고, 4 이상은 최근 3roudn 이내에 best cost 가 나왔을 경우 일찍 종료될 수도 있다. 실험 결과를 보면, round 4 까지는 성능이 너무 나빠져서 사용하기 힘들고, round 5 나 6 정도가 실제 사용할 수 있는 값으로 보인다. 그러나, round6 일 경우에는 SAD 연산량이 Random Access 기준 5.8%이상 증가하는데 반해 round5 일 경우는 0.06%의 성능저하만으로 15%이상의 SAD 연산량이 감소하므로 round5 를 선택하는 것이 적절해 보인다. 이 상태에서 raster search 알고리즘에 대한 조건을 적용하도록 한다.

4.3.3. Raster Search

표 5 TZS 세부 알고리즘에 따른 SAD연산량 비중

Random Access				Lowdelay-B			
Grid	Raster	Star	Etc	Grid	Raster	Star	Etc
35.3%	46.3%	16.0%	2.3%	28.0%	52.6%	17.7%	1.8%

(1920x1080 32frames)

Raster search 는 grid search 에서 찾은 best distance 의 크기가 5 보다 클 경우 실행되므로, 실행빈도가 큰 편은 아니다. 그러나 Raster search 의 search 영역이 ±64 이고 해당 범위 내에서 5 pixel 간격으로 sampling 하여 SAD 연산을 수행하므로, 한 번 raster search 를 수행할 때마다 기본적으로 (129/5)x(129/5)=675 번의 SAD 연산을 수행한다. 이런 이유로 적은 실행빈도에도 불구하고 표 5 에 나타낸 바와 같이 TZS 알고리즘에서 발생하는 SAD 연산의 절반 가량을 차지하고 있다. 그러므로, 이 부분에 대한

중복률을 높이는 것이 reuse buffer 의 사용률을 높이는 데 있어서 중요한 요인이 된다.

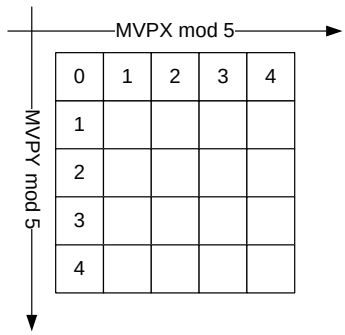
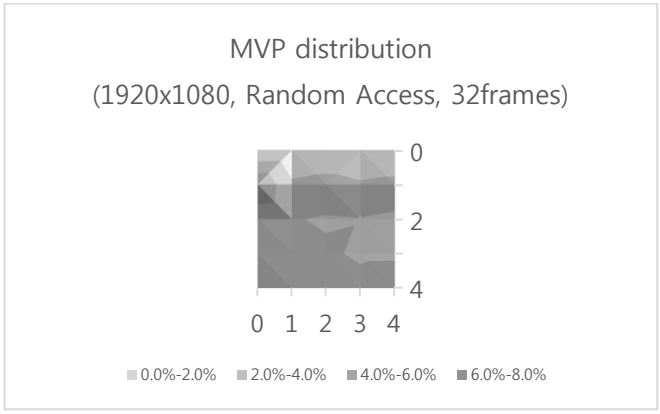


그림 19 Raster search의 start position의 예

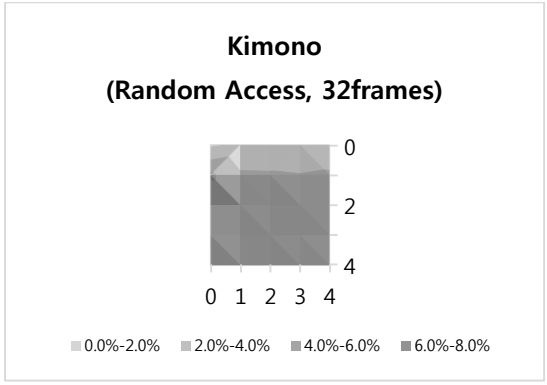
그림 18 에서 raster search 의 중복률이 대략 60%임을 확인하였는데, 중복률을 높이기 위해 Raster search 의 start position 의 위치를 조정하는 방법에 대해 실험을 진행하였다. Raster search 는 5pixel 간격으로 sampling 하여 SAD 연산을 수행하므로, 그림 19 처럼 x 축 5pixel 과 y 축 5pixel 영역에 대해 총 25 개의 서로 다른 initial motion vector position 이 존재하게 된다. Raster search 는 특정 영역 이내가 아닌 search range 전체에 대해서 찾는 것이기 때문에 1~2pixel 정도의 오차는 성능에 크게 영향을 주지 않을 것으로 예상된다. 게다가, raster search 이후 star refinement 를 통해서 best position 을 더 세밀하게 찾기 때문에 여기에서 생긴 오차가 어느 정도는 보정이 가능할 것으로 기대된다. 그러므로, 25 개의 raster search set 중에서 특정 set 하나를 선택하여 raster search 를 진행하도록 하면 중복률을 높여서 더 많은 데이터를 재사용할 수 있을 것으로 보인다. 이를 위해, TZS search 를 시작할 때 주어지는 motion vector

predictior 의 위치가 5x5 영역의 어느 지점에서 발생하는지 분포를 확인하였다.

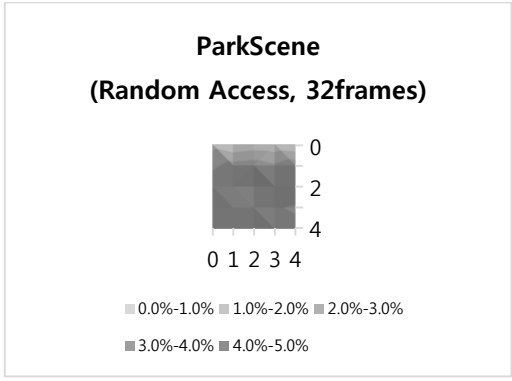


	0	1	2	3	4
0	2.1 %	2.1 %	2.3 %	2.2 %	2.1 %
1	8.0 %	4.3 %	4.8 %	4.2 %	4.6 %
2	4.7 %	4.1 %	3.9 %	4.0 %	3.8 %
3	4.6 %	4.3 %	4.1 %	3.9 %	3.9 %
4	4.4 %	4.6 %	4.5 %	4.3 %	4.5 %

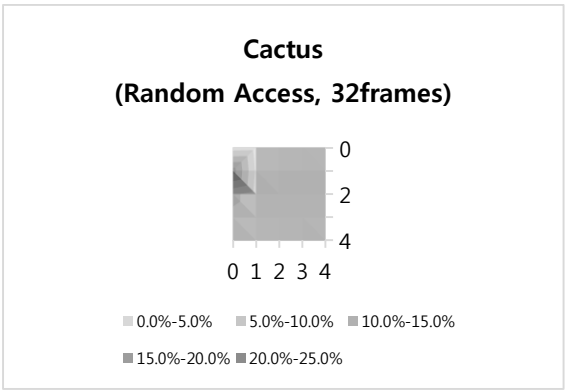
(a) MVP distribution (1920x1080 images)

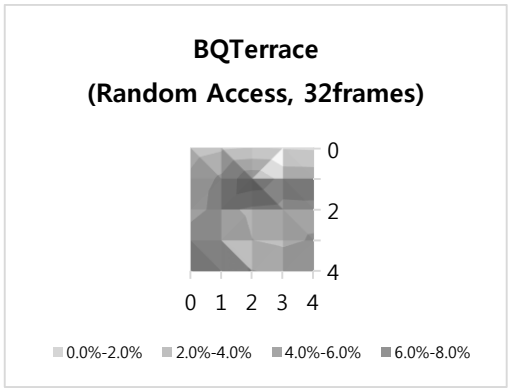


(b) MVP distribution(Kimono)



(c) MVP distribution (ParkScene)





(d) MVP distribution (Cactus) (e) MVP Distribution (BQTerrace)

그림 20 5x5 영역에 대한 Motion Vector Predictor의 발생 분포

(1920x1080 이미지, Random Access, 32frames)

그림 20 은 raster search 를 수행하는 간격인 5pixel x 5pixel 범위 내에서 Motion Vector Predictor 가 발생하는 분포를 조사한 것이다. 그림 20 의 (a)는 1920x1080 이미지인 Kimono, ParkScene, Cactus, BasketballDrive, BQTerrace 5 개 시퀀스에 대해 각각 32frame 씩 수행하여 데이터를 수집한 것에 대해 평균을 취한 것이며, (b)~(e)는 개별 시퀀스에 대한 분포를 표시한 것이다. 그림에서 짙은 색 부분이 발생 확률이 높음을 의미한다. 이미지 시퀀스마다 motion vector 의 특성이 다르기 때문에 발생 분포는 이미지 종류에 의존적이기는 하지만, 전체적으로 보면 대략 (x 좌표,y 좌표)=(0,1)인 지점에서 좀 더 많은 발생 분포를 보이고 있다. 그러므로, 이 지점을 기준으로 하여 raster search 의 start position 을 alignment 하여 실험을 진행하였다.

표 6 Raster Search의 start position을 (0,1)로 align 시켰을 때의 BD-rate

1080p	Random Access				Lowdelay-B			
	6 : 1 : 1	Y	U	V	6 : 1 : 1	Y	U	V
Kimono	0.12%	0.06%	0.60%	0.00%	0.11%	0.09%	0.37%	-0.09%
ParkScene	0.05%	-0.07%	-0.19%	0.18%	-0.04%	-0.01%	0.02%	-0.34%
Cactus	0.01%	-0.01%	0.21%	-0.04%	-0.03%	-0.04%	0.28%	-0.26%
BasketballDrive	0.39%	0.32%	0.52%	0.72%	0.04%	0.08%	0.04%	-0.22%
BQTerrace	-0.13%	-0.18%	-0.26%	0.28%	-0.07%	-0.04%	-0.26%	-0.07%
Average	0.09%	0.05%	0.18%	0.23%	0.00%	0.02%	0.09%	-0.20%

(grid search round=5, star refinement round=4 로 고정)

표 6 의 결과는 위에서 정한 조건인 (x,y)=(0,1)으로 MVP 를 정렬했을 때의 BD-rate 이며, 이 때의 조건은 grid search 의 round 를 5, star refinement 의 round 를 4 로 고정했을 때의 결과이다. Random Access 기준 0.09%가 나빠지는데, 이것은 original TZS 알고리즘 기준 대비 그런 것이고, grid search 와 star refinement 에 대한 조건만을 적용했던 표 4 과 비교하면 0.03% 나빠진 것이므로, raster search 에서의 MVP alignment 는 BD-rate 에 큰 영향을 주지 않는다고 볼 수 있다.

표 7 TZS 알고리즘 실행 조건에 따른 성능 변화 및 SAD연산량 증감률

	Random Access					Lowdelay-B				
	6 : 1 : 1	Y	U	V	SAD 증감률	6 : 1 : 1	Y	U	V	SAD 증감률
[case 1]	0.06%	0.04%	0.15%	0.07%	-10.3%	0.02%	0.03%	0.17%	-0.16%	-11.0%
[case 2]	0.06%	0.05%	0.20%	0.00%	-15.0%	-0.08%	-0.04%	-0.19%	-0.24%	-19.3%
[case 3]	0.09%	0.07%	-0.01%	0.28%	-17.1%	0.00%	0.05%	-0.15%	-0.18%	-21.3%

[case 1] Grid & Raster & (Star with R=4)
[case 2] (Grid with R=5) & Raster & (Star with R=4)
[case 3] (Grid with R=5) & (Raster with alignment) & (Star with R=4)

이제, 위에서 찾은 조건들을 종합하여 알고리즘 실행 조건에 따른 성능 변화 및 SAD 연산량 증감률을 정리하여 표 7 에 표시하였다. 실험의 기준점은 original TZS 알고리즘을 적용하였을 경우이다. [case 1]은 star refinement 의 round 만 4 로 고정하였을 경우로, random access 일 때 BD-rate 은 0.06%가 나빠지는 반면 SAD 연산량은 약 10%정도 감소한다. 이는 star refinement 의 round 제한으로 이 부분의 연산량이 감소하면서 나타나는 현상인데, 표 7 의 [case 1]의 TZS 세부 알고리즘 비중에서 star refinement 의 비중이 16%에서 6.9%로 감소하는 것으로 확인할 수 있다. [case 1] 조건에 grid search 의 round 를 5 로 제한한 [case 2]의 경우를 보면, BD-rate 은 동일한 반면 SAD 연산량이 [case 1] 대비 5% 정도 더 감소하였다. 이 결과는 round 별로 실험을 진행한 표 4 으로부터 얻은 결과인데, round 를 5 로 제한하였더라도, 4round 나 5round 에서 best cost 가 나와서 탐색이 멈추는 경우가 발생하므로 큰 성능저하 없이 SAD 연산량을 감소시킬 수 있었다. [case 2]에 raster search 를 (0,1) 로 align 한 [case 3]의 결과를 보면, BD-rate 은 random access 의 경우 0.09% 나빠지고 lowdelay-B 일 경우 성능저하는 없다. 그리고, 이때의 SAD 증감률은 각각 -17.1%와 -21.3%이다.

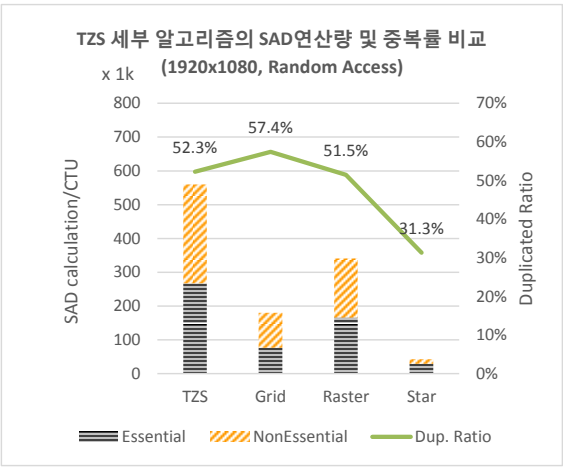
표 8 TZS 세부 알고리즘 실행 조건에 따른 SAD연산량 비중

	Random Access				Lowdelay-B			
	Grid	Raster	Star	Etc	Grid	Raster	Star	Etc

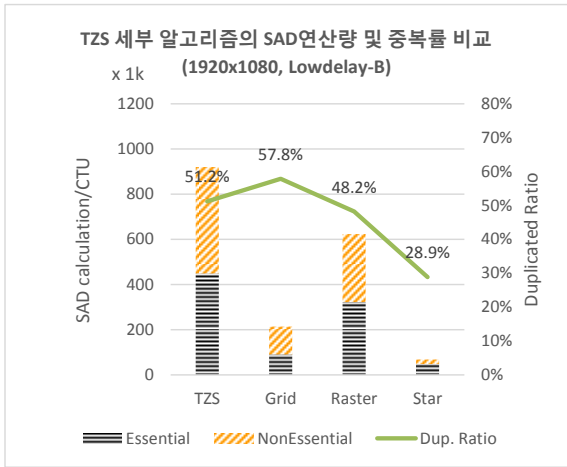
Org	35.3%	46.3%	16.0%	2.3%	28.0%	52.6%	17.7%	1.8%
[case 1]	38.9%	51.7%	6.9%	2.4%	31.2%	59.1%	7.8%	2.0%
[case 2]	28.2%	61.4%	7.9%	2.5%	22.7%	68.5%	7.5%	1.9%
[case 3]	29.1%	60.9%	7.6%	2.4%	23.2%	67.7%	7.4%	1.7%

(1920x1080 32frame, 각 항목은 수정된 알고리즘의 조건을 표시함)

또한, 각 조건에 따른 TZS 세부 알고리즘의 SAD 연산량의 비중도 살펴보았다. Grid search 에서 SAD 연산이 많이 줄어든 만큼 grid search 의 비중이 많이 줄어들었고, 상대적으로 raster search 의 비중이 높아졌다.



(a) Random Access



(b) Lowdelay-B

그림 21 수정된 TZS 세부 알고리즘의 중복률 및 SAD 연산량

(조건:[case 3], 1920x1080)

그림 21 은 수정된 TZS 세부 알고리즘의 중복률 및 SAD 연산량을 표시한 그래프이다. Original TZS 알고리즘 대비 SAD 연산량 17.1% 줄어들었으며, 더불어 중복률 또한 43.7%에서 52.3%로 올라갔음을 확인할 수 있다.

4.4 SAD sub sampling

HM 에는 encoding 을 빠르게 하기 위한 여러 종류의 tool 들이 있는데, SAD 연산 관련해서는 FEN(Fast ENcoding) 설정이 있다. 이 옵션은 HM 에서 기본적으로 설정하게 되어 있는데, 이 옵션을 설정하면 row 의 크기가 8 보다 큰 PU 들에 대해서는 SAD 연산 수행 시 한 줄씩 건너 뛰면서 수행하고 그 값을 2 배 하여 합산하는 sub-sampling[16]방식으로 SAD 연산을 줄이고 있다. 그러므로, 8x8, 8x16, 8x4 등의 PU 들은 정상적으로 모든 pixel 에 대해 SAD 연산을 수행하고, 16x16, 16x8, 32x32, 64x64 등의 크기가 큰 PU 들에 대해서는 한 줄씩 건너 뛰면서 SAD 연산을 수행하는 대신 그 결과를 2 배 하여 합산하는 방식으로 근사화시켜 SAD 연산을 줄이게 된다.

이 방법은 인코딩 성능에 큰 영향을 주지 않으면서도 속도 향상에는 도움이 되는 기능이지만, SAD 연산을 재사용하기 위해 reuse buffer 를 사용하고자 할 때는 문제의 소지가 있다. 왜냐하면 하위 CU depth 의 SAD 연산 결과값을 합산하여 상위 depth 에 대한 reuse buffer 에 저장한 후, 상위 CU depth 에서 그 값을 재사용하는 것이 SAD 연산 재사용을 위한 reuse buffer 의 기본 개념인데, 상위 depth 에서는 sub sampling 한 후 2 를 곱한 값으로 SAD 값을 구하고 있기 때문에 SAD 값이 차이가 나게 되기 때문이다.

또한, reuse buffer 에서 SAD 값을 재사용할 경우 절약한 SAD 연산량을 계산할 때 sub sampling 여부에 따라 sub sampling 을 하지 않았으면 4x4 SAD 연산을 기준으로 1, 했으면 0.5 만큼 더해 주어야 하는데, reuse buffer 에 저장되어 있는 값이 sub sampling 을 하여 저장한 값인지 아니면 SAD 를

그대로 계산한 값인 지 알기 힘들기 때문에 절약한 SAD 연산량을 오차 없이 계산하기에는 reuse buffer 의 구조상 무리가 있다. 그러므로, 본 논문에서는 실험 시 절약한 4x4 단위 SAD 연산량을 계산할 때, row 크기가 8 보다 큰 PU 를 저장하는 reuse buffer 로부터 재사용하면 절약한 SAD 계산 횟수를 0.5 로 하고 row 크기가 8 이하인 PU 를 저장하는 reuse buffer 로부터 재사용하면 절약한 SAD 계산 횟수를 1 로 하였다.

4.5 SAD Data Reuse Process

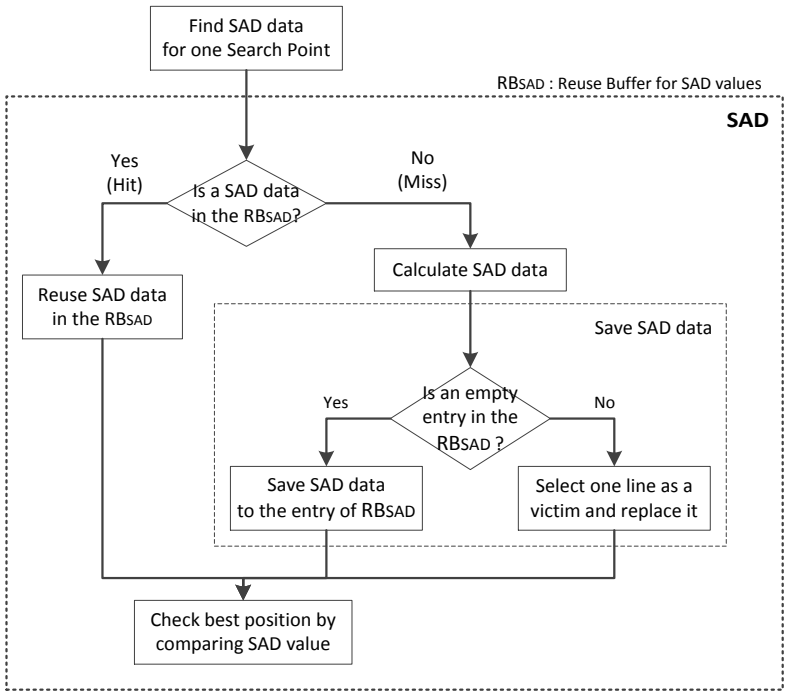


그림 22 SAD data reuse flow

그림 22 에 SAD data 를 재사용하는 flow 에 대해 나타내었다. IME 과정에서 4x4 block 에 대한 SAD 데이터가 RB_{SAD} 에 있으면(hit case) RB_{SAD} 에서 데이터를 가져와서 best position 을 찾는 연산에 활용한다. 만약

데이터가 없으면(miss case) SAD data 를 계산한 후, RB_{SAD} 에 저장하도록 한다. 이 때, 비어있는 entry 가 있으면 그냥 저장하면 되지만, 비어 있는 entry 가 없다면 LRU(least recently used)방법으로 victim line 을 선택한 후 해당 라인을 새로운 값으로 교체하도록 한다.

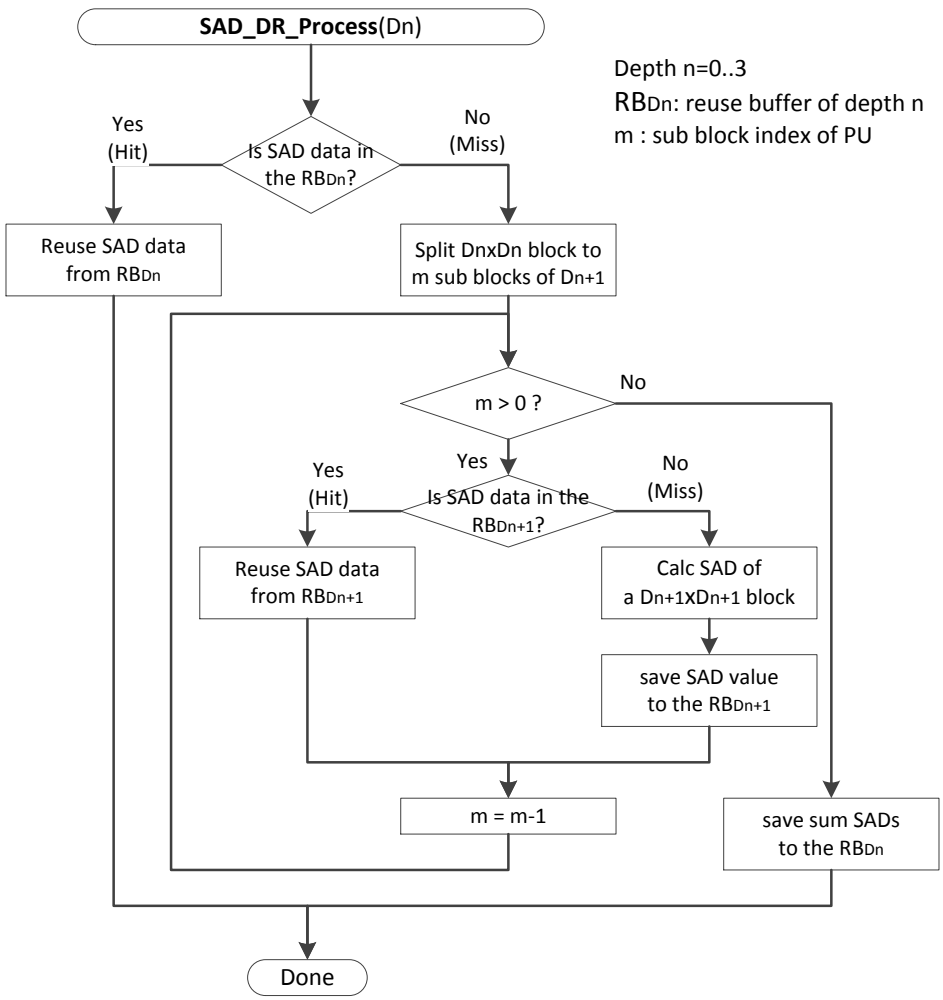


그림 23 SAD Data Reuse flow in detail

그림 23 은 그림 22 에서 reuse buffer 에 SAD 데이터가 없어서 실제로 계산하는 과정인 'Calculate SAD data' 부분을 좀 더 상세하게 표현한

것이다. CU depth 마다 하는 내용은 유사하므로 depth 의 값을 인수로 하여 함수를 호출하도록 한다.

ME 단계에서 특정 크기의 PU 에 대한 SAD 값을 얻고자 한다면, 주소 정보에 있는 tag 와 index, 그리고 block offset 을 사용하여 해당 depth 에 대한 reuse buffer(RB_{Dn}, n=0,1,2,3,4) 로부터 그 값을 가져올 수 있다. CU depth n 에 대해서, 그 정보가 RB_{Dn} 에 있다면(hit case), SAD 값은 별도의 계산 없이 얻을 수 있다. 만약 그 정보가 RB_{Dn} 에 없으면(miss case), PU 를 한 단계 낮은 depth 의 m 개의 2Nx2N PU 들로 나누어서 하위 단계의 RB_{Dn+1} 에서 SAD 값을 찾는다. 만약 여기에 찾는 값이 있으면 이를 재사용하여 상위 단계의 PU 계산에 사용하면 된다. 그러나 만약 여기에도 없다면, 이에 대한 SAD 계산을 수행해야 한다. 이 때, 하위 단계의 reuse buffer 에는 이미 그 아래 단계에서 사용했던 SAD 값이 있을 것이기 때문에 2 단계 낮은 reuse buffer 에서 SAD 값을 찾을 필요는 없다. 이렇게 계산한 SAD 값은 하위 단계의 reuse buffer 를 채우면서 동시에 상위 depth 의 PU 에 대한 SAD 를 계산하는 것에 사용된다.

제 5 장 SATD Data Reuse 알고리즘

5.1 SATD 데이터의 중복률

표 9 Redundant SATD calculation ratio by 4x4 block unit considering pipeline structure

Sequences	Half SATD		Quarter SATD	
	Random Access	Lowdelay-B	Random Access	Lowdelay-B
Kimono	21.0%	19.8%	16.5%	14.8%
ParkScene	30.8%	29.3%	27.0%	25.3%
Cactus	31.5%	30.3%	28.5%	27.5%
BasketballDrive	21.5%	19.3%	18.0%	16.0%
BQTerrace	32.5%	31.3%	29.8%	28.5%
Average	27.5%	26.0%	24.0%	22.4%

(32frames, QP=22,27,32,37)

SATD 연산 중에서 현재 얼마나 많은 중복 연산이 이루어지고 있는지 파악하기 위해 HM 12.0 의 common test condition 환경에서 이를 조사하였다. 표 9 은 중복된 SATD 연산에 대해 각 이미지 sequence 별로 나타낸 것이다. 원본 이미지의 PU 를 각각 4x4 block 단위로 분할하여 이들 각각에 대해 참조 이미지의 데이터로부터 SATD 를 계산하였다. 참조하는 이미지도 같고 원본 block 과 예측 block 의 위치가 같은 4x4 block 에 대한 SATD 값을 이미 계산한 적이 있으면, 그 이후부터는 중복인 것으로 처리하였으며, 중복률 R_{DUP}는 다음과 같은 식으로 구하였다.

$$R_{DUP} = \frac{Duplicated\ SATD\ Counts}{(Essential\ SATD\ Counts + Duplicated\ SATD\ Counts)}$$

표 9 에 이러한 과정으로 구한 중복률을 표시하였으며, Full HD 시퀀스 5 개에 대해 적용해 본 결과 Random Access 의 경우 Half SATD 계산의 약 27.5%, Quarter SATD 계산의 약 24.0%가 중복 연산임을 알 수 있었다.

5.2 SATD Data Reuse Process

SATD 데이터를 재사용하는 흐름은 '4.5 SAD Data Reuse Process'와 유사하다. 그러나, SATD 값을 구하려면 sub-pixel 을 interpolation 을 통해 구해야 하고, sub-pixel 또한 half-pixel 과 quarter-pixel 로 되어 있으므로, 이들 사이에 데이터를 reuse 하는 process 를 잘 정의해야 한다.

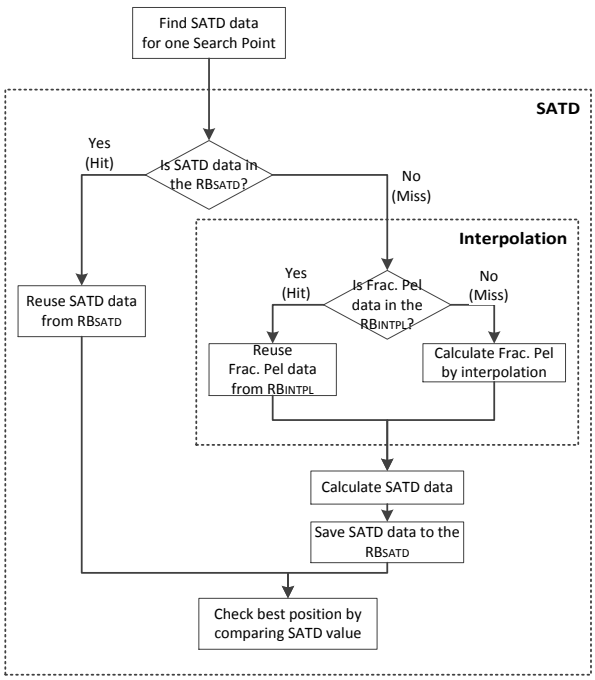


그림 24 SATD & interpolation data reuse flow

그림 24 은 FME 에서 fractional search point 에 대해서 pixel 을 생성하고 SATD 를 구할 때, reuse buffer 를 사용하여 기존 연산 값을 재사용하는 흐름을 보여주고 있다. 먼저, integer motion vector 주변 8 개 중 하나의 search point 에 대한 SATD 값을 구하고자 할 때, 먼저 SATD 값에 대한 reuse buffer 인 RB_{SATD} 에 해당 값이 있는지 체크한다. 그 값이 있으면(hit) 가져와서 사용하면 되지만, 그 값이 없으면(miss) SATD 값을 계산해야 한다. 이 때, SATD 값을 계산하기 위해서는 sub pixel 정보가 있어야 하므로, 다시 interpolation 값에 대한 reuse buffer 인 RB_{INTPL} 에 해당 값이 있는지 체크한다. 그 값이 있으면(hit) 가져와서 사용하면 되지만, 그 값이 없으면(miss) interpolation 값을 계산하여 사용하도록 한다. 이렇게 구한 sub pixel 값을 사용하여 SATD 값을 구하게 되는데, 새로 구한 SATD 값은 다시 RB_{SATD} 에 저장하여 나중에 같은 곳에 대한 SATD 요구가 있을 때 활용하도록 한다.

Sub pixel 은 half-pixel 과 quarter-pixel 이 있는데, 이들 pixel 좌표는 서로 겹치지 않으므로 같은 reuse buffer 를 사용한다고 해도 중복되는 값은 발생하지 않는다. 그러므로, half-pixel 용 reuse buffer 와 quarter-pixel 용 reuse buffer 로 분리하여 중복 데이터를 재사용하도록 한다.

제 6 장 Interpolation Data Reuse 알고리즘

그림 24 에서 언급하였듯이, interpolation 은 FME 과정에서 fractional MV 를 구하기 위해 sub-pixel 에 대한 SATD 값을 계산하기 위해 필요한 값이다. 7-tap 혹은 8-tap 에 대해 곱셈과 덧셈연산을 하기 때문에 interpolation 연산 자체의 복잡도도 상당하지만, PU partition mode 마다 이러한 과정이 반복되기 때문에 중복되는 연산이 매우 많다. 게다가, current picture 의 pixel 과 계산하여야 하는 SAD 나 SATD 와는 달리 reference picture 의 pixel 만으로 연산이 가능하기 때문에 중복성이 더 높다.

이러한 이유로 interpolation data 에 대한 reuse buffer 는 cache 구조가 아닌 단순 buffer 형태로 하며, FME 전에 CTU 단위로 미리 계산하여 저장한 후, FME 과정에서 중복되는 데이터가 있으면 재사용하는 방식으로 구성하였다. 이 때, 중복되지 않아서 새로 계산한 interpolation data 는 reuse buffer 에 저장하지 않으며, 나중에 그 값이 다시 필요하면 새로 계산해야 하는 부담이 있다. 그러므로, CTU 범위만큼 interpolation 을 미리 할 때 초기 위치를 정확히 선정하는 것이 재사용 율을 높이는 중요한 요소가 된다.

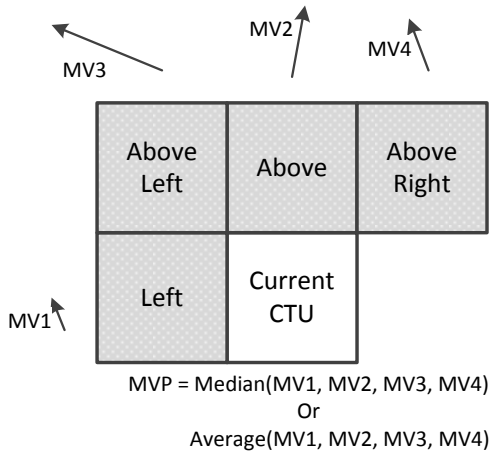


그림 25 interpolation영역의 조정을 위한 median vector 생성

그림 25 는 interpolation 하는 영역을 효과적으로 설정하기 위하여 주변 PU 의 motion vector 를 이용하는 방법을 나타내고 있다. 현재 CTU 주변에 있는 available 한 PU 의 motion vector 의 median 값을 구하여 이를 reference frame 에 있는 CTU 에 적용하면, 보정 없이 CTU 에 대해 interpolation 을 수행하는 것보다 데이터의 재사용 율을 높일 수 있다.

제 7 장 Reuse Buffer 구조

7.1 Reuse Buffer Architecture for SAD/SATD

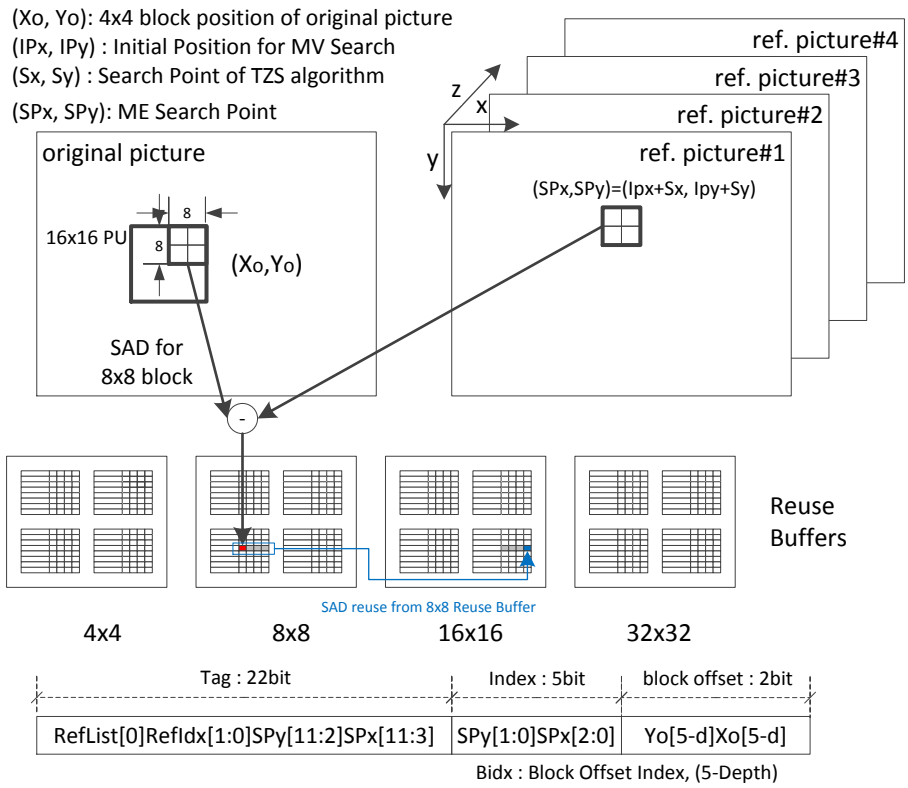


그림 26 SAD 및 SATD 데이터 재사용을 위한 Reuse Buffer

그림 26 는 tag 및 SAD data array 로 이루어져 있는 reuse buffer 를 나타내고 있다. Tag 는 reference list 정보, reference index 정보 및 reference picture 에 대한 search point 좌표의 일부를 가지고 있다. Tag 는 Reuse buffer 에 SAD 값을 저장하거나 필요한 SAD 값을 가져올 때 사용하는 정보이다. Index 는 n-way set associative 로 reuse buffer 를 구성할 때, 버퍼의 특정 위치를 찾아갈 때 필요한 값으로 search point 좌표 일부를

가지고 있다. SAD data array 는 8Byte 로 구성되어 있으며, 4x4 에 대한 reuse buffer 의 경우 한 줄에 4x4 block 에 대한 SAD 값 4 개를 가지고 있다. 8x8, 16x16, 그리고 32x32 reuse buffer 또한 한 줄에 각각 8x8 block 에 대한 SAD 값 4 개, 16x16 reuse buffer 에 대한 SAD 값 4 개, 그리고 32x32 reuse buffer 에 대한 SAD 값 4 개를 가지게 된다. 단, 64x64 reuse buffer 만 하나의 라인에 한 개의 64x64 SAD 값만 가지게 된다. 한 줄에 존재하는 4 개의 SAD 값을 선택하는 것은 Block offset 으로 사용되는 original picture position 의 좌표인데, original picture 에서 x, y 좌표 1bit 씩 가져와서 구성하고 있다. Address 의 Index 부분은 reuse buffer 의 특정 set 을 찾아갈 때 사용되고, tag 부분은 선택된 set 에서 특정 way 를 찾을 때 사용되며, block offset 은 선택된 한 라인에 존재하는 4 개의 SAD data 중 하나를 선택할 때 사용된다.

위 구조에서 reuse buffer 의 크기를 효율적으로 관리하기 위해 생각해 볼 것들이 있다. SAD 값은 original picture 의 CTU 좌표와 reference picture 의 좌표정보가 모두 있어야만 정확히 계산할 수 있다. 그러나 그림 26 의 아래쪽의 address 구성을 보면, tag 에 original picture 의 좌표가 없는 것을 확인할 수 있다. 이는 HM 의 inter prediction 단계에서 CU 단위로 motion vector 를 찾을 때 그림 27 에서처럼 zig-zag scan 방식으로 탐색한다는 사실 및 reuse buffer 가 depth 별로 구성되어 있다는 사실과 관련이 있다. 하위 depth 4 개의 CU 에 대해 Zig-zag scan 을 수행할 때, CU 4 개에 속한 PU 들에 대해서 TZS search 가 수행되며, 이 과정에서 발생하는 SAD 값들이 해당 depth 에 대한 reuse buffer 에 저장되고 재사용된다. 이 과정이 끝나면 상위 depth 에 대해서 탐색을 하게 되는데, 상위 depth 와 하위 depth 의

CTU 내에서의 위치가 동일하고 search point 좌표가 동일하면 하위 depth 의 reuse buffer 에 있는 SAD 값을 재사용할 수 있다. 만약 하위 depth 에 재사용할 SAD 값이 없다면 새로 계산하여 사용한다. 상위 depth 의 zig-zag scan 위치가 옮겨지면, 하위 depth 의 reuse buffer 에 있는 내용은 더 이상 사용하지 않게 되므로, 이 시점에서 buffer 를 비우고 새로운 값을 채우게 된다. 하위 depth 에 대한 reuse buffer 는 상위 depth 에서 위치를 변경하면서 계속 재사용되므로, 이들의 위치에 해당하는 값을 tag 에 넣을 필요는 없다. 대신, 현재 어느 위치에 대한 계산을 하고 있는지에 대한 정보는 내부 state machine 에서 관리하면 될 것이다. 이런 과정은 depth 3, 2, 1 에 대해 모두 동일하며, depth0 는 CTU 위치가 변하지 않으므로 역시 tag 에 CTU 위치 정보가 있을 필요가 없고, block offset 정보 또한 가지고 있을 필요가 없다.

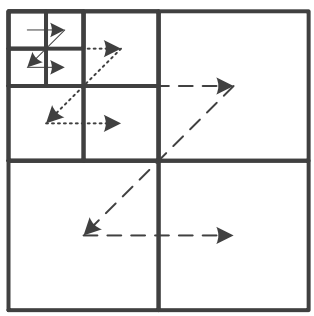


그림 27 HM에서의 CU에 대한 zig-zag 탐색 순서

이제 이렇게 구성된 reuse buffer 의 동작에 대해 살펴보자. Inter prediction 의 flow 를 작은 크기의 CU 부터 진행하는 것으로 변경하였으므로, 8x8 block 에 대해 재사용하는 구조를 살펴보자. 그림 26 은 각 PU depth 의 SAD 값을 저장하는 buffer 구조를 나타내고 있다. 그림에서는 8x8 block 에 대해 SAD 값을 구하는 경우와 16x16 block 에 대해 SAD 를 구하는 과정을

나타내고 있다. 8x8 block 에 대한 SAD 값을 구할 때, 먼저 8x8 reuse buffer 를 살펴본다. 만약 있으면 바로 사용하면 되지만, 없을 경우에는 다시 4x4 reuse buffer 를 살펴본다. 4x4 reuse buffer 의 해당 라인에 SAD 데이터 4 개가 있으면 가져와서 사용하면 되고, 없으면 새로 계산해서 8x8 block 에 대한 SAD 값을 구하여 8x8 reuse buffer 에 저장하고 사용한다. 여기에서 새로 계산한 4x4 SAD 값은 4x4 reuse buffer 에서 저장한다. 16x16 block 에 대한 SAD 연산과정 역시 유사하다. 먼저 16x16 reuse buffer 에서 찾아보고 있으면 사용하고 없으면 8x8 reuse buffer 에서 찾아본다. 여기에 있으면 사용하면 되고, 없을 경우 4x4 block 단위로 계산하여 사용하게 된다. 이 때 계산한 4x4 SAD 값은 4x4 reuse buffer 에 저장되며 이들 값을 합산하여 다시 8x8 reuse buffer 및 16x16 reuse buffer 에도 저장한다.

앞에서 기술하였듯이 이렇게 저장된 값은 특정 영역에서의 상위 depth 에 대한 ME 연산이 끝나면 지워질 수 있다. 예를 들어, (0,0)기준으로 8x8 범위에 있는 4 개의 4x4 block 에 대한 SAD 값은, 이들 4 개에 대한 ME 연산 후 8x8 연산 끝난 다음 지울 수 있으며, 이러한 방법으로 reuse buffer 의 크기를 절약할 수 있다.

7.2 Reuse Buffer Architecture for interpolation

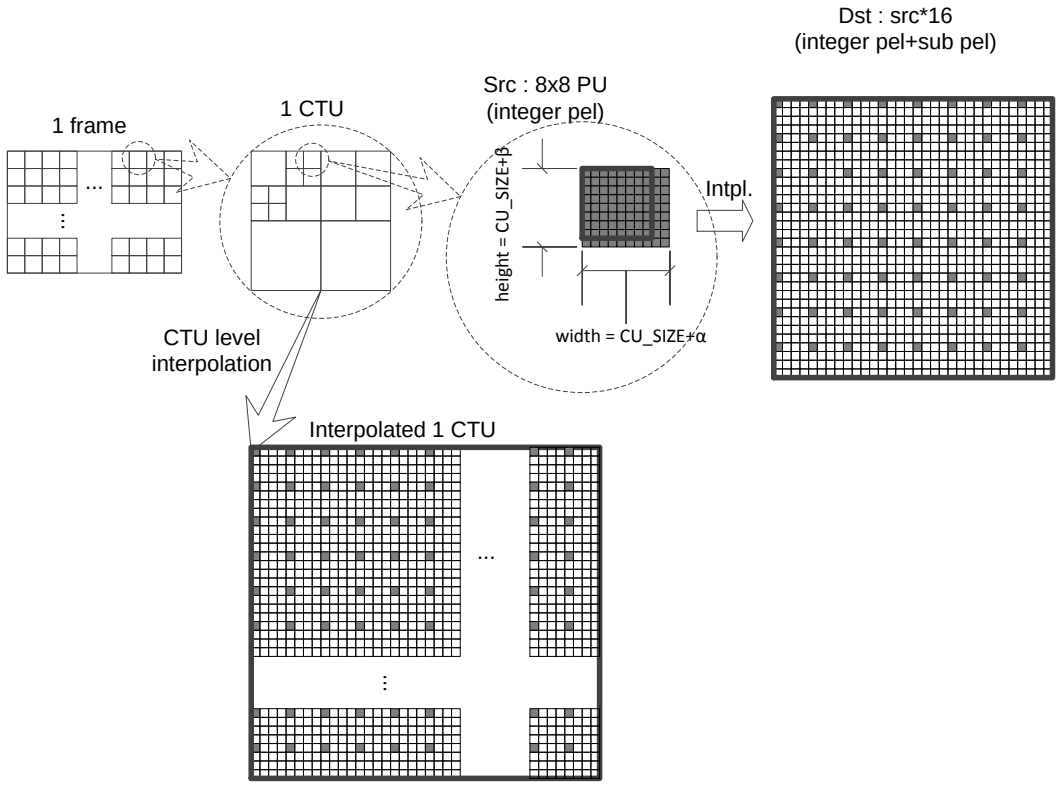


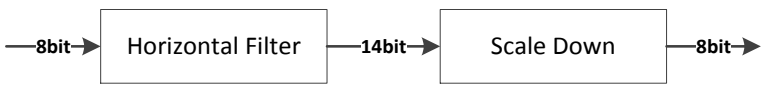
그림 28 interpolation reuse buffer for CTU level data reuse

그림 28 은 CTU 단위로 미리 계산하여 저장하는 interpolation reuse buffer 를 나타내고 있다. 그림에 표현된 CTU 를 보면, CTU 영역 전체에 대한 interpolation 을 수행하여 buffer 에 저장하는 부분이 CTU 아래쪽으로 표시되어 있고, CTU 내의 PU 하나에 대해서 prediction 단계에서 PU 크기만큼 interpolation 을 수행하는 것을 CTU 그림 오른쪽에서 보여주고 있다. 그러므로, reference picture 마다 CTU 영역에 대해 interpolation 을

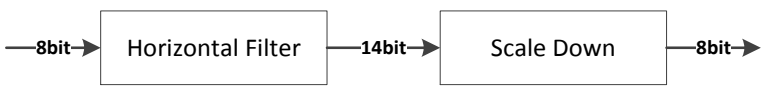
수행하여 저장하면, ME 단계에서 CTU 내의 모든 CU 및 모든 PU partition 에 대한 interpolation 연산시 그 결과가 buffer 에 이미 계산되어 있으면 데이터를 가져와서 재사용할 수 있다.

interpolation 으로 인해 변화하는 정밀도(bit depth)는 filtering 연산의 종류에 따라 다르지만, 최종적으로는 interpolation 하기 전의 값들과의 SATD 비교를 위해서 그림 29 처럼 scale down 하여 8bit 으로 맞춰 주어야 한다.

(a) 가로와 세로 필터가 모두 적용되는 경우



(b) 가로 필터가 적용되는 경우



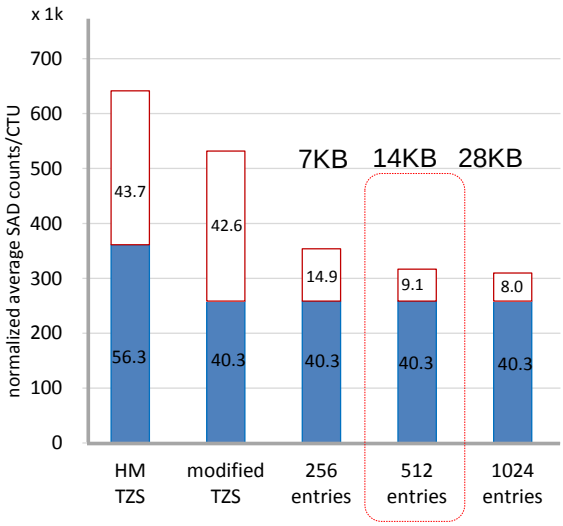
(c) 세로 필터가 적용되는 경우

그림 29 interpolation filter에 따른 정밀도

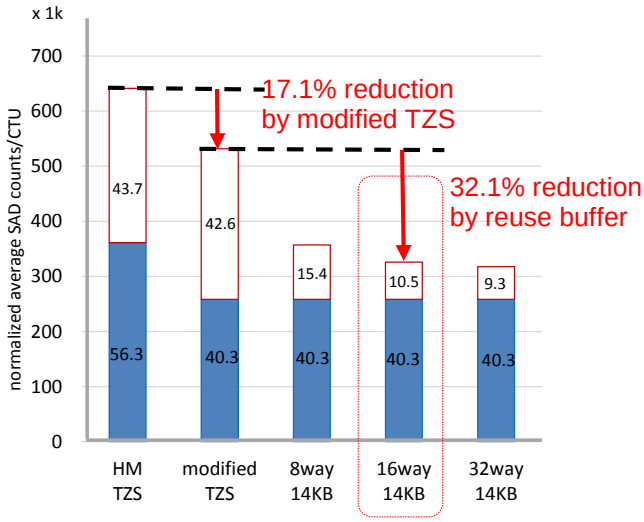
ME 단계에서 필요한 것은 search algorithm 으로 찾은 fractional mv 의 8bit 짜리 pixel 값이므로, filtering 연산을 수행하여 저장하는 값은 필터를 곱하여 scale-up 된 값이 아니라 scale-down 되어 기존 pixel 과의 bit 길이가 같은 값이다. 그러므로, 하나의 값을 저장하기 위해서는 1Byte 가 필요하다. 다음은 64x64 CTU 영역에 대한 interpolation 데이터를 저장하기 위해 필요한 메모리 크기이며, reference picture 하나 당 (64x4)x(64x4)x1Bytes=64KB 이다.

제 8 장 실험 결과

8.1 SAD 실험 결과



(a) Fully associative reuse buffer



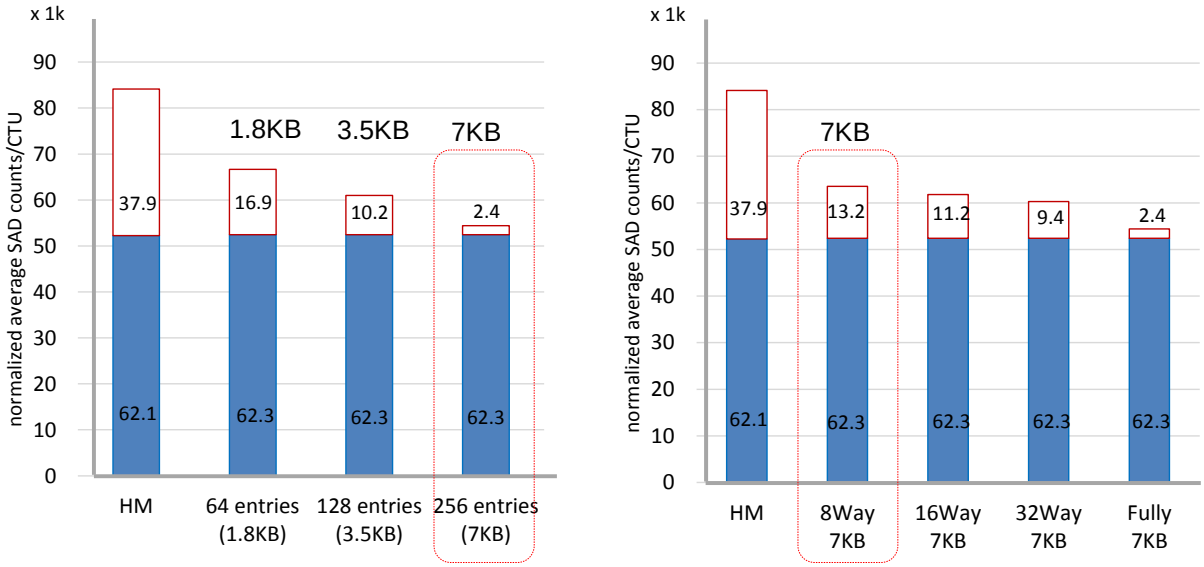
(b) n-way set associative reuse buffer

그림 30 modified TZS를 사용하여 IME를 수행하였을 때의 SAD 연산 복잡도 비교

그림 30 은 pipeline 구조를 고려하여 수정된 TZS 알고리즘으로 reuse buffer 를 사용하였을 때의 SAD 연산량이 얼마만큼 감소 되었는지 나타낸 결과이다. 'HM TZS' 는 HM encoder 의 TZS 알고리즘을 사용하여 IME 를 수행한 SAD 연산량을, 'modified TZS' 는 '4.3 TZS algorithm modification' 항목의 내용을 적용하여 수정한 TZS 알고리즘을 사용하여 IME 를 수행한 SAD 연산량을 의미하며, 이 때에는 reuse buffer 를 사용하지 않았다. 세로축은 하나의 CTU 에 대해 4x4 block 단위 SAD 로 정규화한 SAD 의 평균 연산 복잡도를 의미한다. 둘로 나뉘어진 막대그래프의 아래

부분과 윗부분은 각각 ‘필수적인 SAD 연산량’과 ‘중복된 SAD 연산량’을 의미하며, 막대 그래프 안쪽의 숫자는 ‘HM TZS’ 인 경우 대비 SAD 연산량을 의미한다. 그림 30 (a) 그래프는 reuse buffer 를 fully associative 로 구현하였을 때의 결과로, 각 depth 별 reuse buffer 를 모두 동일하게 256, 512, 1024 entry 로 설정했을 때의 SAD 연산량을 의미한다. 이 때 중복 SAD 연산량은 ‘HM TZS’ 인 경우의 43.7% 대비 각각 14.9%, 9.1%, 8.0%로 감소하였음을 알 수 있다. 모든 depth 의 Reuse buffer 들의 entry 가 모두 512 인 경우, 수정된 TZS 알고리즘을 사용한 IME 연산으로 인한 SAD 연산량은 ‘HM TZS’ 인 경우 대비 49.4%로 줄었음을 확인할 수 있다. 그리고, 이 때의 SAD data 를 위한 reuse buffer 의 크기는 14KB 이다. 그림 30 (b) 그래프는 14KB 인 reuse buffer 에 대해 set associative 의 way 에 따른 SAD 연산 복잡도를 비교한 것이다. 8way, 16way, 32way 인 경우에 대해 각각 55.7%, 50.8%, 49.6%로 fully associative 일 때인 49.4%와 차이가 있는 것을 확인할 수 있다. 이는 modified TZS 알고리즘을 사용하여 줄인 17.1%와 16way set associative reuse buffer 20KB 를 사용하여 줄인 32.1%를 포함하여 TZS 로 인한 SAD 연산량의 49.2%를 줄일 수 있음을 의미한다. TZS 로 인한 SAD 연산량은 bi-prediction 까지 포함한 전체 SAD 연산량과 비교하면 70% 정도이므로, 전체 SAD 연산량의 34.4%를 줄일 수 있다.

8.2 SATD 실험 결과



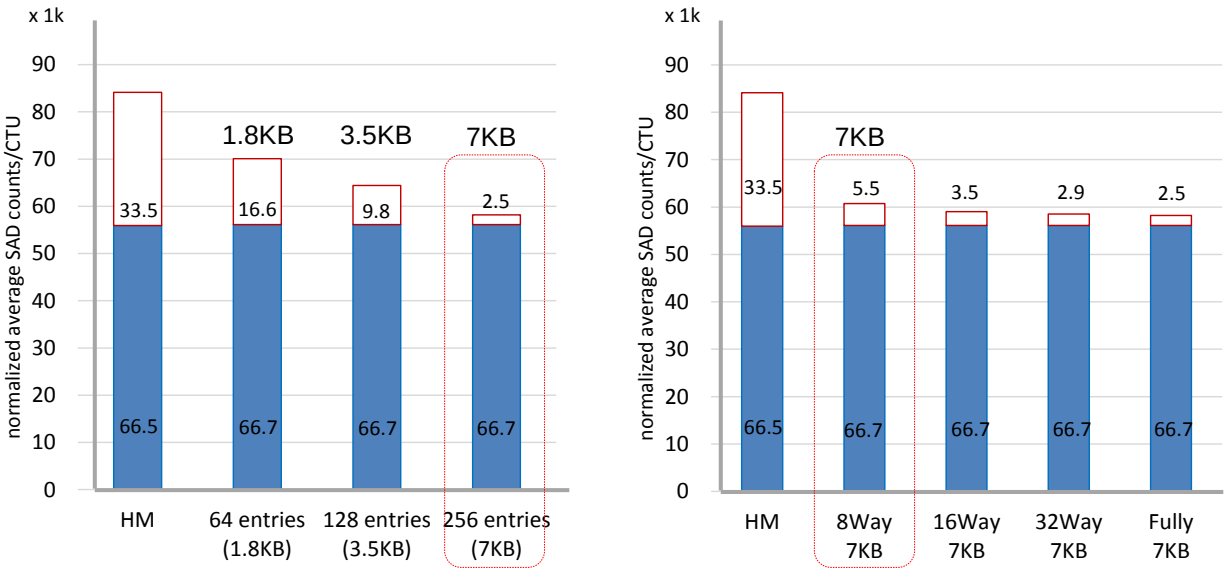
(a) Fully associative reuse buffer

(b) n-way set associative reuse buffer

그림 31 FME를 수행하였을 때의 half SATD 연산 복잡도 비교

그림 31 은 half pel 에 대한 SATD 연산 결과에 reuse buffer 를 사용하였을 때의 half SATD 연산량이 얼마만큼 감소 되었는지 나타낸 결과이다. 'HM' 은 HM encoder 의 FME 에서 half pel 에 대한 half SATD 를 수행하였을 때의 연산량 의미하며, 이 때에는 reuse buffer 를 사용하지 않았다. 세로축은 하나의 CTU 에 대해 4x4 block 단위 half SATD 로 정규화한 half SATD 의 평균 연산 복잡도를 의미한다. 둘로 나뉘어진 막대그래프의 아래 부분과 윗부분은 각각 '필수적인 half SATD 연산량' 과 '중복된 half SATD 연산량' 을 의미하며, 막대 그래프 안쪽의 숫자는 'HM' 인 경우 대비 half SATD 연산량을 의미한다. 그림 31 (a) 그래프는 reuse

buffer 를 fully associative 로 구현하였을 때의 결과로, 각 depth 별 reuse buffer 를 모두 동일하게 64, 128, 256 entry 로 설정했을 때의 half SATD 연산량을 의미한다. 이 때 중복 half SATD 연산량은 'HM' 인 경우의 37.9% 대비 각각 16.9%, 10.2%, 2.4%로 감소하였음을 알 수 있다. 모든 depth 의 Reuse buffer 들의 entry 가 모두 256 인 경우, half SATD 연산량은 'HM' 인 경우 대비 64.7%로 줄었음을 확인할 수 있다. 그리고, 이 때의 half SATD data 를 위한 reuse buffer 의 크기는 7KB 이다. 그림 31 (b) 그래프는 7KB 인 reuse buffer 에 대해 set associative 의 way 에 따른 half SATD 연산 복잡도를 비교한 것이다. 8way, 16way, 32way 인 경우에 대해 각각 75.5%, 73.5%, 71.7%로 fully associative 일 때인 64.7% 와 7~11% 정도의 차이가 있는 것을 확인할 수 있다. 이는 8way set associative reuse buffer 를 사용할 때, 7KB SRAM 을 사용하여 half SATD 연산량의 24.5%를 줄일 수 있음을 의미한다.



(a) Fully associative reuse buffer (b) n-way set associative reuse buffer

그림 32 FME를 수행하였을 때의 quarter SATD 연산 복잡도 비교

그림 32 은 quarter pel 에 대한 SATD 연산 결과에 reuse buffer 를 사용하였을 때의 quarter SATD 연산량이 얼마만큼 감소 되었는지 나타낸 결과이다. 'HM' 은 HM encoder 의 FME 에서 quarter pel 에 대한 quarter SATD 를 수행하였을 때의 연산량 의미하며, 이 때에는 reuse buffer 를 사용하지 않았다. 세로축은 하나의 CTU 에 대해 4x4 block 단위 quarter SATD 로 정규화한 quarter SATD 의 평균 연산 복잡도를 의미한다. 둘로 나뉘어진 막대그래프의 아래 부분과 윗부분은 각각 '필수적인 quarter SATD 연산량' 과 '중복된 quarter SATD 연산량' 을 의미하며, 막대 그래프 안쪽의 숫자는 'HM' 인 경우 대비 quarter SATD 연산량을 의미한다. 그림 32 (a) 그래프는 reuse buffer 를 fully associative 로 구현하였을 때의 결과로, 각 depth 별 reuse buffer 를 모두 동일하게 64, 128,

256 entry 로 설정했을 때의 quarter SATD 연산량을 의미한다. 이 때 중복 quarter SATD 연산량은 'HM' 인 경우의 33.5% 대비 각각 16.6%, 9.8%, 2.5%로 감소하였음을 알 수 있다. 모든 depth 의 Reuse buffer 들의 entry 가 모두 256 인 경우, quarter SATD 연산량은 'HM' 인 경우 대비 69.2%로 줄었음을 확인할 수 있다. 그리고, 이 때의 quarter SATD data 를 위한 reuse buffer 의 크기는 7KB 이다. 그림 32 (b) 그래프는 7KB 인 reuse buffer 에 대해 set associative 의 way 에 따른 quarter SATD 연산 복잡도를 비교한 것이다. 8way, 16way, 32way 인 경우에 대해 각각 72.2%, 70.2%, 69.6%로 fully associative 일 때인 69.2% 와 0.4~3% 정도의 차이가 있는 것을 확인할 수 있다. 이는 8way set associative reuse buffer 를 사용할 때, 7KB SRAM 을 사용하여 quarter SATD 연산량의 27.8%를 줄일 수 있음을 의미한다.

종합하면, half SATD 와 quarter SATD 연산량에 대해서 총 14KB SRAM 을 사용하여 8way set associative 로 이루어진 reuse buffer 를 사용하면, 26.2%의 연산량을 절약할 수 있다. 그리고, 이 결과는 uni-prediction 으로 인한 것만 생각한 것이므로, bi-prediction 까지 포함한 전체 연산결과 대비 절약할 수 있는 SATD 연산량은 18.3%이다.

8.3 Interpolation 실험 결과

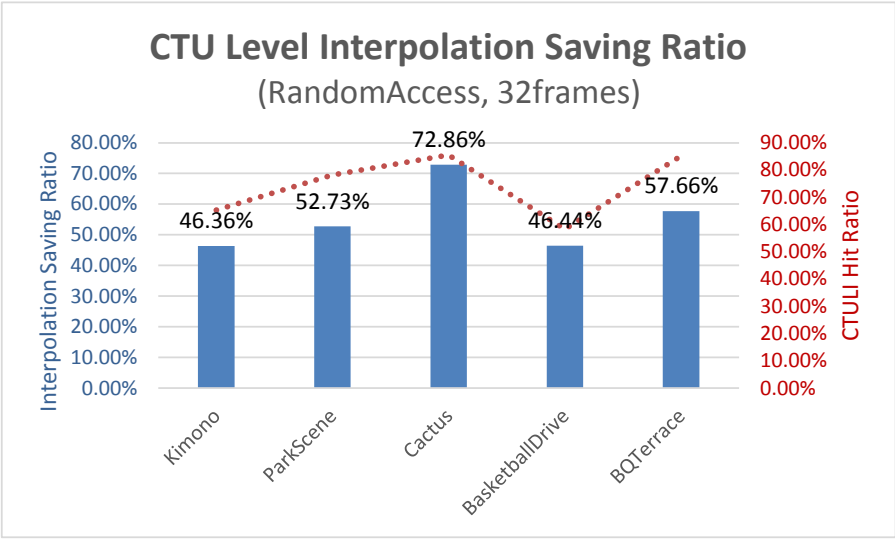


그림 33 CTU Level로 interpolation data를 재사용하였을 경우의 기존 대비 계산량 절약 비율

그림 33 는 CTU Level 로 interpolation 을 미리 수행한 후, ME 연산 단계에서 interpolation 데이터를 재사용하였을 때 절약 가능한 interpolation 의 연산을 표시한 그림이다. Image sequence 의 motion vector 특성에 따라 재사용할 수 있는 interpolation 의 양이 차이가 나지만, 대략 50%정도의 재사용률을 보이고 있다.

8.4 Data Reuse Throughput

8.4.1 Design Target

Reuse buffer 를 적용하기 위한 H/W design specification 은 다음과 같다.

- UHD(4096x2160) 30fps @ 400MHz
- Supports 4 ref. pics and 7 PU partition modes
- Supports modified TZS algorithm with 100 search points per each PU

8.4.2 Throughput calculation

위에서 설정한 조건 아래에서 CTU 당 할당되는 cycle 및 이를 만족하는 parallelism 을 결정하기 위하여 [17] 논문에 있는 다음과 같은 식을 이용한다.

$$\text{cycles assigned per CTU} = \frac{frequency}{CTU\ rate}$$

$$= \frac{frequency * 64 * 64}{frame\_width * frame\_height * frame\_rate} \quad (1)$$

$$\text{parallelis m needed} = \frac{predicted\ samples}{cycles\ assigned\ per\ CTU * utilization\ n} \quad (2)$$

Target design 은 400MHz 로 4Kx2K 30fps 인코딩이 가능한 HEVC 인코더이다. 이러한 조건을 이용하여 (1) 식으로 CTU 당 IME 에서 필요한 cycle 을 구하면 다음과 같다.

$$\frac{400MHz * 64 * 64}{4096 * 2160 * 30} = 6173\ cycles$$

그러므로, h/w를 구성할 때 이 조건을 만족할 수 있도록 (2) 식을 이용하되, 계산상의 편의와 utilization margin을 고려하여 6000 cycle로 계산하도록 한다.

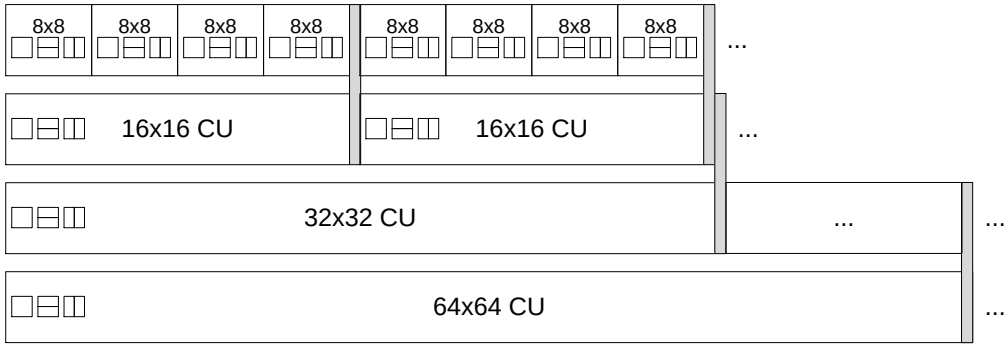


그림 34 CU depth 별로 SAD unit 이 할당되어 CU depth 에 대해 병렬적으로 처리하는 경우에 대한 PU mode 결정 과정

그림 34은 각 CU depth별로 SAD unit이 할당되어 CU마다 따로 계산하는 경우의 H/W 구조이다. 각 CU depth별로는 PU partition에 대한 SAD계산이 순차적으로 처리되므로 이들 사이의 SAD data reuse가 발생하지만, CU depth 사이의 data reuse는 발생하지 않는다. 여기에 앞에서 정한 design의 throughput인 6000cycles/CTU를 맞추기 위해서 SAD 연산 유닛을 병렬로 처리하도록 구성하도록 한다. 이에 대해 다음 절에서 살펴보도록 한다.

8.4.3 SAD cycle 계산

표 10 SAD 연산에 대한 cycle 계산

			TZS		Modified TZS with reuse buffer		
level	calc. SAD (cycle)	Part. modes	Search points of TZS	w/o reuse buffer (cycle)	Search points of modified TZS	reuse ratio	w/ reuse buffer (cycle)
8x8	4	3	100	307200	83	41.2%	176188
16x16	16	7		716800			365147
32x32	64	7		716800			353657
64x64	256	7		716800			350785
Total(cycle)	–	–	–	2457600	–	–	1245777

[condition] avg. # of TZS search points are 100, # of PU partitions is 7, and # of ref. pictures is 4.

표 10은 그림 34 의 H/W 구조를 고려하여 각 CU depth 별로 계산한 cycle 수이다. 먼저 16x16 CU에 대해서 살펴보면, CTU 하나에 할당된 cycle이 6000cycle이므로, 총 16개인 16x16 CU 하나를 처리하기 위해 할당된 cycle은 6000/16=375 cycle 이다. 4x4 SAD unit이 1 cycle에 처리되므로 16x16 SAD는 16 cycle이 걸리고, 4장의 reference picture와 7개의 PU partition 그리고 100개의 TZS search point를 고려하여 필요한 cycle을 계산하면, 16(cycles)x4(refs.)x7(partitions)x100(search points) = 44800 cycles 이 된다. 16x16 CU 하나에 대한 cycle budget은 375cycle이므로, 100% utilization을 고려해도 최소 120개의 4x4 SAD unit이 있어야 성능을 만족할 수 있다. 이는 cycle이 동일하게 할당된 32x32나 64x64 CU depth도 마찬가지이다. 다만, 8x8 CU depth의 경우, PU partition이 3종류이므로 이 경우에만 52개의 SAD 이 필요하므로, 모든 CU depth에 대해 총 412개의 SAD unit이 필요하다.

이 때, SAD 데이터의 재사용 여부를 살펴보자. CU depth 사이는 병렬 수행하므로 SAD데이터를 재사용하기 힘들며, PU partition들 사이의 재사용이 가능하다. PU partition들 사이의 재사용이 가능하려면, throughput을 맞추기 위해 CU depth당 할당된 120개의 SAD 엔진으로 하나의 partition에 대한 계산을 수행해야만 그 다음 partition에서 그 결과를 다시 사용할 수 있다. TZS search의 3 round 의 20개의 search point 사이에는 중복된 SAD연산이 없으므로 이들을 동시에 처리하더라도 연산량 재사용에는 지장이 없다. 8x8 CU depth의 경우, 4(cycles)x4(refs.)x20(search points)=320 이 되어 52개의

SAD엔진으로 한 번에 모두 처리할 수 없으므로, 하나의 partition에 대한 계산 결과는 추후 다른 partition에서 재사용될 수 있다. 이러한 내용은 다른 CU depth에서도 마찬가지이다.

Modified TZS 알고리즘을 사용하면, SAD연산량이 17% 줄어들게 되고, 이들 연산량에 reuse buffer 를 적용하면 modified TZS로 인해 줄어든 연산량의 41.2%(원래의 SAD연산량과 비교하면 32.1%)의 연산량이 추가적으로 줄어들게 되어 총 49.2%의 SAD연산량이 줄어들게 된다. 이를 바탕으로 필요한 cycle을 계산하면, 표 10 에서처럼 각각 2457600cycle과 1245777cycle이 필요하며, 6000cycle/CTU 의 budget을 만족시키기 위해서는 각각 410개와 208개의 SAD unit이 필요하다. 이를 바탕으로 SAD unit의 H/W complexity를 비교하면 20KB의 reuse buffer를 사용하여 49.2%를 줄일 수 있다.

8.4.4 SATD cycle 계산

표 11 SATD 연산에 대한 cycle 계산

level	calc. SAD (cycle)	reuse SAD (cycle)	reuse ratio	w/o reuse buffer (cycle)	w/ reuse buffer (cycle)
8x8	4	1	23%	55296	45757
16x16	16	1	23%	129024	101203
32x32	64	1	22%	129024	101082
64x64	256	1	22%	129024	100750
Total(cycle)	–	–	–	442368	348792

[condition] avg. # of search points are 18, # of PU partitions is 7, and # of ref. pictures is 4.

표 11은 그림 34 의 H/W 구조를 고려하여 각 CU depth 별로 계산한

cycle 수이다. 먼저 16x16 CU에 대해서 살펴보면, CTU 하나에 할당된 cycle이 6000cycle이므로, 총 16개인 16x16 CU 하나를 처리하기 위해 할당된 cycle은 6000/16=375 cycle 이다. 4x4 SATD unit이 1 cycle에 처리되는 것으로 가정하면 16x16 SATD는 16 cycle이 걸리고, 4장의 reference picture와 7개의 PU partition, 그리고 18개의 search point를 고려하여 필요한 cycle을 계산하면, 16(cycles)x4(refs.)x7(partitions)x18(search points) = 8064 cycles 이 된다. 16x16 CU 하나에 대한 cycle budget은 375cycle이므로, 100% utilization을 고려하면 최소 22개의 4x4 SATD unit이 있어야 성능을 만족할 수 있다. 이는 cycle이 동일하게 할당된 다른 CU depth도 마찬가지이다. 다만, 8x8 CU depth의 경우, PU partition이 3종류로 이 경우에만 10개의 SATD unit이 필요하므로, 모든 CU depth에 대해 총 76개의 SATD unit이 필요하다.

이 때, SATD 데이터의 재사용 여부를 살펴보자. CU depth 사이는 병렬 수행하므로 SATD데이터를 재사용하기 힘들며, PU partition들 사이의 재사용이 가능하다. PU partition 들 사이의 재사용이 가능하려면, CU depth당 할당된 22개의 SATD엔진으로 한 번에 계산하는 SATD연산의 수가 하나의 partition내에서 동시에 계산 가능한 수보다 적어야만 그 다음 partition에서 이전 partition의 SATD값을 재사용할 수 있다. 8x8 CU의 경우, 4(cycles)x4(refs.)x18(search points)=288 이 되어 10개의 SATD엔진으로 한 번에 모두 처리할 수 없다. 그러므로, 하나의 PU partition에 대한 계산 결과는 추후 다른 PU partition에서 재사용될 수 있다. 이러한 결과는 다른 depth에 대해서도 마찬가지이다.

표 11에서 확인할 수 있듯이, reuse buffer를 사용하면 사용하지 않을 때보

다 21.2%의 cycle을 절약할 수 있다. 이 때 필요한 reuse buffer는 half SATD와 quarter SATD 합산하여 20KB 이다.

제 9 장 결론

본 논문에서는 inter mode decision 과정에서 PU 단위로 motion vector 를 구하기 위해 수행되는 SAD, SATD 및 interpolation 연산이 HEVC 인코더 연산양의 절반 이상을 차지하는 점에 착안하여 연산 복잡도를 줄이는 것을 연구의 방향으로 삼았다. 이를 위해 PU partition 모드마다 반복적으로 수행되는 SAD, SATD 및 interpolation 을 reuse buffer 에 저장한 후, ME 과정에서 같은 값을 요구할 때 가져와서 재사용하는 방법을 사용하였다.

데이터의 중복성 및 절약한 연산량을 정량적으로 파악하기 위하여, 다양한 PU 의 크기를 4x4 크기로 환원하여 이에 대한 SAD, SATD 및 interpolation 의 실제 수행 횟수를 파악하였다. 그리고, 이 기준에 맞추어 각 연산 종류별로 중복률을 파악하였다.

HM 에서 사용하는 TZS 알고리즘은 grid-raster-star search 순서로 수행되며 search point 를 찾는 방법이 각각 다르기 때문에, TZS 알고리즘을 그대로 사용하면 reuse buffer 의 활용에 제약이 있다. 때문에 data reuse 에 적합하도록 TZS algorithm 을 일부 수정하여 데이터의 재사용성을 높였다.

중복 데이터를 저장하기 위해서는 on-chip memory buffer 가 필요한데, 이를 reuse buffer 라고 하였다. Reuse buffer 의 크기와 데이터 재사용율 사이에는 trade-off 관계가 있는데, buffer 의 크기가 무한정 클 수는 없으므로, 적절한 크기로 데이터의 재사용율을 높이도록 해야 한다.

SAD 와 SATD 연산은 작은 크기의 block 에 대해 계산한 값을 더하여 더 큰 block 의 연산값을 구할 때 합산하여 사용할 수 있다. 이러한 성질을 이용하기 위하여 CU depth 별로 reuse buffer 를 따로 마련하였으며, 하위

단계의 결과를 상위 단계의 reuse buffer 에 저장한 후 하위 단계의 reuse buffer 는 초기화하여 사용하는 방식으로 reuse buffer 의 크기를 작게 사용하였다. Interpolation 의 경우, 이러한 특성은 없지만 current picture 의 값과는 상관없이 reference picture 의 pixel 만을 사용하므로, CU depth 구분없이 reference picture 별로 CTU 단위에 대한 interpolation 결과를 저장하여 재사용하는 방식으로 하였다.

그 결과, tag 와 data 를 포함한 20KB 의 on-chip 메모리를 사용하여 전체 SAD 연산량의 약 34.4%, 20KB 의 on-chip 메모리를 사용하여 전체 SATD 연산량의 약 18.3%, 그리고 256KB 의 메모리를 사용하여 전체 interpolation 연산량의 약 50%를 절약하는 결과를 얻었다. 이 때의 성능 저하는 modified TZS algorithm 으로 인해 0.09%와 pipeline 구조로 인해 수정된 ME 의 초기조건으로 인해 0.26%를 포함한 0.35%이다. 이는 SAD, SATD 및 interpolation 연산량이 전체 인코더의 약 50% 이상 차지하는 것을 고려할 때, 제안한 reuse buffer 를 사용하면 전체 인코더의 computational complexity 를 약 18%이상 줄일 수 있음을 의미한다.

참고 문헌

[1] G. Correa, P. Assuncao, L. Agostini, and L. Cruz, "Coding tree depth estimation for complexity reduction of HEVC," in *Data Compression Conference (DCC), 2013*, 2013, pp. 43-52.

[2] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "HEVC complexity and implementation analysis," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 22, pp. 1685-1696, 2012.

[3] Y.-H. Chen, T.-C. Chen, C.-Y. Tsai, S.-F. Tsai, and L.-G. Chen, "Data reuse exploration for low power motion estimation architecture design in H. 264 encoder," *Journal of Signal Processing Systems,* vol. 50, pp. 1-17, 2008.

[4] T.-C. Chen, Y.-H. Chen, S.-F. Tsai, S.-Y. Chien, and L.-G. Chen, "Fast algorithm and architecture design of low-power integer motion estimation for H. 264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 17, pp. 568-577, 2007.

[5] H. F. Ates and Y. Altunbasak, "SAD reuse in hierarchical motion estimation for the H. 264 encoder," *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05).* 2005, pp. ii/905-ii/908 Vol. 2.

[6] Y.-W. Huang, T.-C. Wang, B.-Y. Hsieh, and L.-G. Chen, "Hardware architecture design for variable block size motion estimation in MPEG-4 AVC/JVT/ITU-T H. 264," *2003. ISCAS'03. Proceedings of the 2003 International Symposium on Circuits and Systems,* 2003, pp. II-796-II-799 vol. 2.

[7] S. Y. Yap and J. V. McCanny, "A VLSI architecture for variable block size video motion estimation," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 51, pp. 384-389, 2004.

[8] C.-Y. Chen, S.-Y. Chien, Y.-W. Huang, T.-C. Chen, T.-C. Wang, and L.-G. Chen, "Analysis and architecture design of variable block-size motion estimation for H. 264/AVC," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 53, pp. 578-593, 2006.

[9] R. Li, B. Zeng, and M. L. Liou, "A new three-step search algorithm for block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 4, pp. 438-442, 1994.

[10] L.-M. Po and W.-C. Ma, "A novel four-step search algorithm for fast block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 6, pp. 313-317, 1996.

[11] J. Y. Tham, S. Ranganath, M. Ranganath, and A. A. Kassim, "A novel unrestricted center-biased diamond search algorithm for block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 8, pp. 369-377, 1998.

[12] N. Purnachand, L. N. Alves, and A. Navarro, "Improvements to TZ search motion estimation algorithm for multiview video coding," *2012 19th International Conference on Systems, Signals and Image Processing (IWSSIP),* 2012, pp. 388-391.

[13] Y. Song, M. Shao, Z. Liu, S. Li, L. Li, T. Ikenaga, *et al.*, "H. 264/AVC fractional motion estimation engine with computation reusing in HDTV1080p real-time encoding applications," *2007 IEEE Workshop on Signal Processing Systems,* 2007, pp. 509-514.

[14] Z. Liu, Y. Song, M. Shao, S. Li, L. Li, S. Ishiwata, *et al.*, "HDTV1080p H. 264/AVC encoder chip design and performance analysis," *IEEE Journal of Solid-State Circuits,* vol. 44, pp. 594-608, 2009.

[15] F. Bossen, "JCTVC-I1100: Common test conditions and software reference configurations," *Joint Collaborative Team on Video Coding (JCT-VC),* 2012.

[16] B. Liu and A. Zaccarin, "New fast algorithms for the estimation of block motion vectors," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 3, pp. 148-157, 1993.

[17] C.-G. Peng, D.-S. Yu, X.-X. Cao, and S.-M. Sheng, "Efficient VLSI design and implementation of integer motion estimation for H. 264 SDTV encoder," *2006. ICSICT'06. 8th International Conference on Solid-State and Integrated Circuit Technology,* 2006, pp. 2019-2021.

Abstract

Reuse Buffer Architecture for Reducing the Computational Complexity of Inter Prediction in HEVC encoder

Roe Sunil

Dept. of Electrical and Computer Engineering

The Graduate School

Seoul National University

Motion Estimation (ME) consists of two parts: integer ME (IME) and fractional ME (FME). Integer ME finds an integer motion vector (IMV) for each prediction unit (PU) and fractional ME (FME) makes fractional pixels and finds a fractional motion vector for each PU.

The initial search point for IME is selected from the AMVP candidate list that are generated only from the available neighbor MVs of the current PU which depends on the pipeline architecture of the HEVC encoders. For IME, the low complexity RD cost based on the sum of absolute difference (SAD) values is calculated for each search point determined by the TZS algorithm and the predicted motion vector is selected for the search point with the minimum RD cost. In FME, the fractional pixels are first constructed through interpolation filtering using 7 or 8 integer pixels and then the fractional motion vector is selected from the search points around

the integer motion vector, which has the minimum RD cost based on the SATD.

In the HM encoder the RDO search algorithm finds a MV for each reference picture for each PU in the allowable PU partitions of every CU in all the allowable CU partitions for each coding tree units (CTU) in the current picture. Therefore there can be more than 1000 motion vectors for bi-prediction to be found for each CTU. To find an integer motion vector, the TZS algorithm can search more than 100 search points on the average. To find a fractional motion vector, the FME algorithm can search 16 points at most. Thus, the computation for SAD, SATD, and interpolation operations occupies a large share in the total computational complexity of the HEVC encoders.

In this paper, we present a modified TZS algorithm to reduce the SAD computation and increase computation redundancy, which make the reuse buffer suitable to reduce computational complexity. We also present a scheme that reduce duplicated computation by pre-calculating those data of reference frames for interpolation. To manage stored data efficiently, we adopt a reuse buffer with cache architecture considering parallel operation of each CU depth especially for SAD and SATD.

As a result, we can get about 34.4% reduction in SAD computation using 20KB on-chip memory, 18.3% reduction in SATD computation using 20KB on-chip memory, and 50% reduction in interpolation computation using 256KB on-chip memory with 0.35% performance degradation including 0.09% caused by modified TZS algorithm and 0.26% by pseudo-AMVP

candidate list. It means that about 18% reduction in the overall computational complexity of encoder with 0.35% performance degradation and 296KB on-chip memory.

Keywords : HEVC, SAD, SATD, Interpolation, Data Reuse

Student Number : 2013-20784